NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ORTHOGONAL WALL FOLLOWING AND OBSTACLE AVOIDANCE BY AN AUTONOMOUS VEHICLE

by

Daniel A. Wells

June 1999

Thesis Advisor: Yutaka Kanayama
Second Reader: Thomas Hofler

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1999	Master's Thesis

4. TITLE AND SUBTITLE
ORTHOGONAL WALL FOLLOWING AND OBSTACLE AVOIDANCE BY AN AUTONOMOUS VEHICLE

5. FUNDING NUMBERS

6. AUTHOR(S)
Wells, Daniel A.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The purpose of this thesis was to integrate a wall following motion mode for a rigid body autonomous vehicle. Yamabico, an autonomous vehicle located at the Naval Postgraduate School, was used as the test and evaluation platform.

To implement the new motion mode, the vehicle was required to follow a straight wall with minor variations, navigate around corners, and avoid obstacles in its path while maintaining a specified offset distance from continuously connected wall segments. Sonar transmitter/receiver pairs were used to sense the environment and collect positional data for analysis. Modifications to pre-existing motion and sensor software libraries on board Yamabico were performed to achieve the motion goals. One of the major contributions from these modifications was the addition of a linear fitting algorithm using a decay factor. The algorithm produced quick response by the vehicle to changing conditions in its environment.

The experimental results by Yamabico were successful with the algorithm developed by the author. The result of this thesis is that an autonomous vehicle can be given the capability to perform smooth and efficient motion adjustments to an environment composed of orthogonal wall segments and obstacles.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Autonomous Vehicles, Wall Following, Obstacle Avoidance	170
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFI- CATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

ORTHOGONAL WALL FOLLOWING AND OBSTACLE AVOIDANCE BY AN AUTONOMOUS VEHICLE

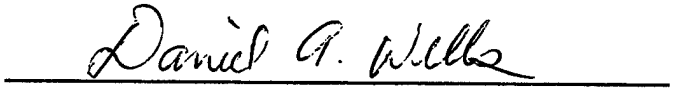
Daniel A. Wells
Lieutenant, United States Navy
B.S., University of Colorado, 1990

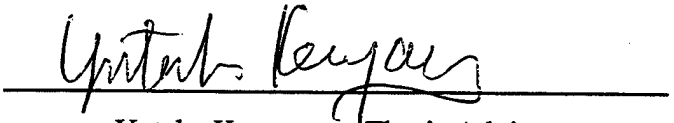
Submitted in partial fulfillment of the
requirements for the degree of

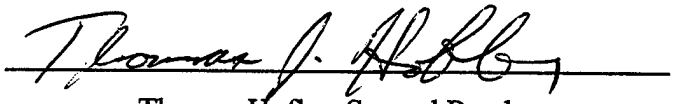
MASTER OF SCIENCE IN COMPUTER SCIENCE

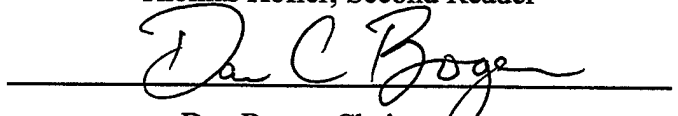
from the

NAVAL POSTGRADUATE SCHOOL
June 1999

Author: 
Daniel A. Wells

Approved by: 
Yutaka Kanayama, Thesis Advisor


Thomas Hofler, Second Reader


Dan Boger, Chairman
Department of Computer Science

ABSTRACT

The purpose of this thesis was to integrate a wall following motion mode for a rigid body autonomous vehicle. Yamabico, an autonomous vehicle located at the Naval Postgraduate School, was used as the test and evaluation platform.

To implement the new motion mode, the vehicle was required to follow a straight wall with minor variations, navigate around corners, and avoid obstacles in its path while maintaining a specified offset distance from continuously connected wall segments. Sonar transmitter/receiver pairs were used to sense the environment and collect positional data for analysis. Modifications to pre-existing motion and sensor software libraries on board Yamabico were performed to achieve the motion goals. One of the major contributions from these modifications was the addition of a linear fitting algorithm using a decay factor. The algorithm produced quick response by the vehicle to changing conditions in its environment.

The experimental results by Yamabico were successful with the algorithm developed by the author. The result of this thesis is that an autonomous vehicle can be given the capability to perform smooth and efficient motion adjustments to an environment composed of orthogonal wall segments and obstacles.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

All biological beings, from humans to insects, share the ability to move about in their world without constantly bumping into obstacles along their path of movement. For insects and less sophisticated animals, this ability is simply a sense and react type of behavior. More sophisticated species, including humans, are able to sense their surroundings with multiple sensors, process and correlate the inputs, and perform complex analyses to determine future movements.

Robotics and artificial intelligence research tries to emulate these behaviors. On the surface, this might appear to be a relatively straightforward task. In reality, even the simplest behaviors can encompass an enormous amount of complexity, and emulating animal behavior in an artificial system is extremely difficult. This fact is evident when considering that this research area has been around for many decades while yielding few major robotics breakthroughs in natural movement abilities. This is not to say that there has been no significant progress. On the contrary, there have been many important contributions by this research area such as smart weapons, unmanned aerial vehicles, and robots sent to explore solar system planets, to name a few. These examples indicate that robotics and artificial intelligence is an important and vital area of research, but at the same time they show that implementing even simple functional behavior, relative to human standards, takes years to research and develop into actual working models.

The examples mentioned in the previous paragraph are considered to have simple functional behavior because they are either preprogrammed to perform specific tasks or are remotely controlled by an operator. More sophisticated models must be able to perform autonomously. To do this, an artificial entity must be able to analyze sensory inputs and make decisions based on that analysis without outside help. This process is something humans do easily and, for the most part, take for granted. On the other hand, implementing this process in hardware and software requires much effort. Dr. Yutaka Kanayama of the Naval Postgraduate School has been working on this problem for the better part of twenty years. In particular, he has created, maintained, and continuously modified a library of movement and sensory algorithms and data structures called the Model-based Mobile robot Language (MML). The robot Yamabico (Figure 1) has been the testing and implementation platform for experimentation with and analysis of MML.

B. OVERVIEW

This thesis is an extension of Dr. Kanayama's work on the Yamabico robot and the MML software library. Yamabico is an autonomous robot capable of using its sensor inputs and controlling its motion based on those inputs. This functionality is provided by programming code written in the C programming language and contained in the numerous files of the MML software library. Yamabico is also capable of following a predefined path using path tracking and rotational motion modes.

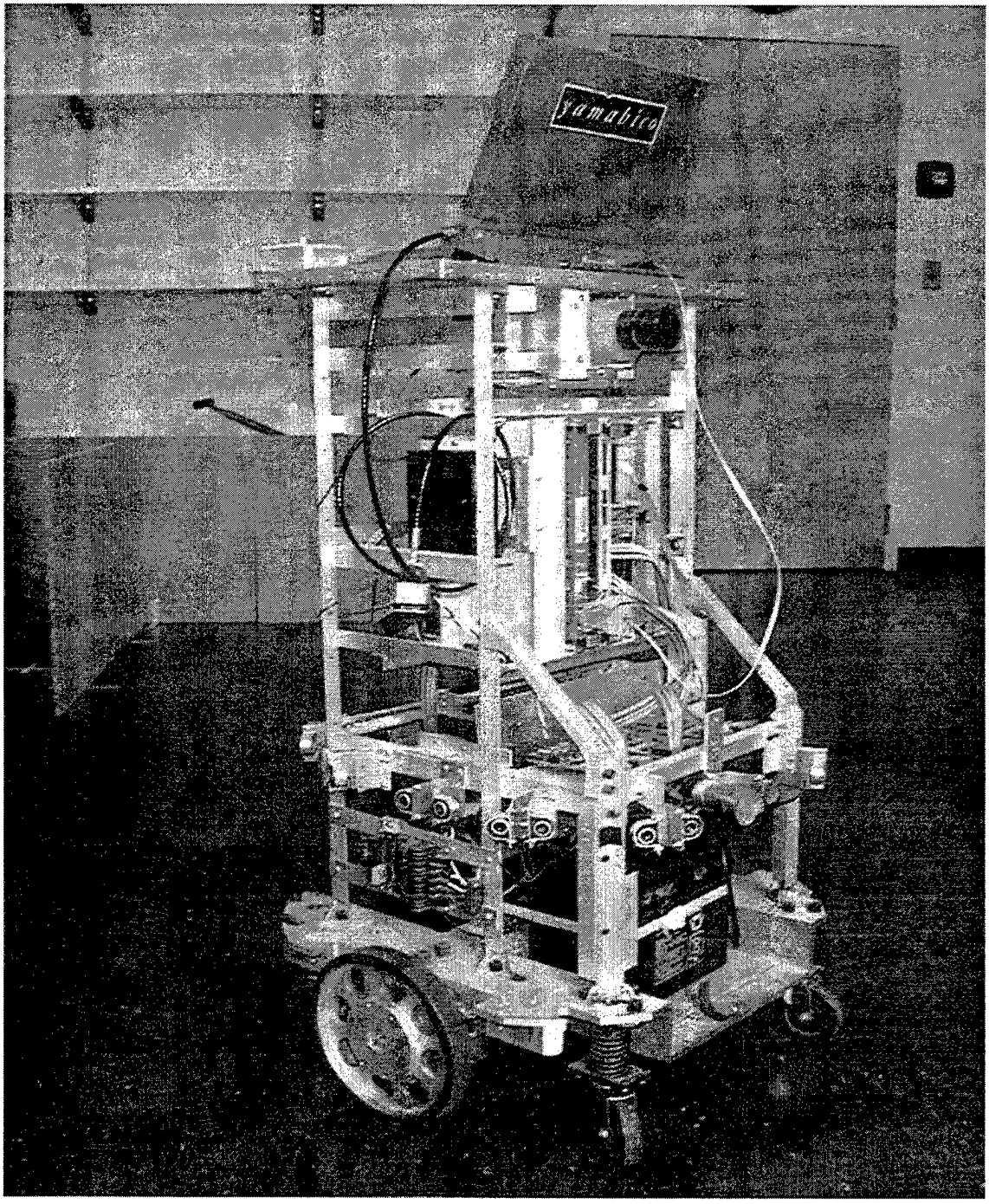


Figure 1. The Autonomous Vehicle Yamabico 11.

In addition to movement capabilities, MML also provides the facility to collect and perform analysis on sensor data. The primary sensor for Yamabico is sonar. Sonar data may be collected by one or more of the twelve sonars arrayed around the perimeter of Yamabico. Each sonar, when enabled, collects data including the distance of the sonar return, the robot's position and orientation at the time of sensing, and the global position of the sonar return. The main analysis technique offered by the sonar system is linear fitting of the collected data points using the least-squares algorithm. Each sonar is capable of maintaining a line representation of its data collection. This ability is important when used for representing objects in the robot's world and maintaining safe distances or defined offsets from those objects.

The tasks Yamabico performs during its run are specified in a user file, also written in C. This file is read in and executed following initialization of the robot. It provides the interface between the robot and the outside world. In this file, the programmer may provide the ability to set initial parameters at run time, specify a path to follow, and/or stipulate how Yamabico should react to particular input events. This is accomplished by using the input and output routines of C and by calling the functions contained in MML.

Yamabico and MML provide extensive and fairly robust motion planning, sensing, and analysis capabilities. These capabilities can be used to direct the robot to perform many different tasks. However, there is currently no inherent ability for the robot to follow a wall or avoid obstacles automatically. These tasks could be programmed into Yamabico via the aforementioned user file, but this would be inefficient. Since the user

file is a type of interface between the outside world and the inner workings of the robot, a level of abstraction is introduced. This means more overhead in the program and increased timing delays between the software and the hardware. These factors can cause significant problems in the robot's performance. It is possible to overcome these problems by careful and exhaustive planning and programming, but it would be much more efficient to give Yamabico the ability to perform these tasks at a lower level (i.e., in the MML software.) These lower level functionalities can be performed by an additional motion mode contained within MML. Implementing this new motion mode in the MML software is the focus of this thesis.

C. PROBLEM STATEMENT

The problem we want to solve in this thesis is to give the robot the capability to make a complete circuit around a room or rooms composed of continuously connected orthogonal wall segments while avoiding obstacles in its path.

To solve this problem, a new wall following motion mode needed to be created.

The first step in realizing this goal was to get the robot to simply follow a straight wall while maintaining a defined offset. The wall was allowed to have minor variations in its "straightness", requiring the robot to make adjustments to its path as it proceeded along the wall.

After this first step was completed, the second step was adding the ability to negotiate orthogonal (90°) turns. To attain this goal, the robot had to handle two types of turns: (1) turns into the robot's current path (concave corners), and (2) turns away from its

path (convex corners). Once this more complex functionality was implemented, all the tools needed by the robot to complete a circuit around a room were in place.

The capability to negotiate obstacles in the robot's path was a side effect of solving the orthogonal wall following problem. Since obstacles in the motion path are in close proximity to the wall being followed, avoiding them was a natural extension of the robot's ability to turn at a wall intersection. By solving the orthogonal turn problem, the requirements to avoid obstacles was also met.

D. ORGANIZATION

Chapter II discusses the hardware characteristics of Yamabico. These include the motion monitoring and control system and the sonar control system.

Chapter III focuses on the software library. The approach used in the software to model physical parameters such as position and movement is somewhat different than what is considered the normal representation. An explanation of the primary characteristics of this model is contained in the first part of the chapter. The second part of the chapter discusses the components of MML relevant to this thesis including the currently implemented motion modes and the linear fitting algorithm.

Chapter IV begins the discussion on thesis specific topics. The programming code related to motion modes is examined in more detail and the modifications required to implement a new wall following mode is explained. A variation of the linear fitting algorithm is discussed and the implication of using such a variation in creating this new motion mode is explained.

Chapter V talks about the results of the experiments with Yamabico and the conclusions reached after the successful implementation of the new motion mode was completed.

II. YAMABICO HARDWARE

All specification data in this chapter was extracted from Reference [1] unless otherwise noted.

A. GENERAL OVERVIEW

The Yamabico robot, formally Yamabico-11, is an autonomous robot composed of several subsystems. The robot stands 90 cm tall with a 50 cm by 50 cm base. The power is supplied by two 12-volt rechargeable batteries that drive the motors, sonars, and processor boards. Yamabico can also be connected to an external power source when stationary to conserve battery power.

A PowerBook™ 145 laptop computer is mounted on top of the robot and is used as a mediator between the outside user, the user program and MML software, and the hardware. All code development and compilation is done on a Sun SPARCstation 10 workstation using the UNIX operating system.

Once a program is compiled, the executable is downloaded through an ethernet connection to a SPARC IV main CPU board on a VME bus. The laptop then runs the program which initializes Yamabico's hardware, executes the user program, and handles the hardware interrupts. After Yamabico has completed its run, any data earmarked for collection via MML logging functions can be downloaded from the laptop to the UNIX network through a standard phone line cable.

There are four major subsystems in Yamabico that interact asynchronously to give it its capabilities. The motion and sonar subsystems are the first two and are the focus of

the remainder of this chapter. The software library and user program comprise the third subsystem. They are discussed in Chapter III. Finally, Yamabico has an optical image processing subsystem, but this subsystem was not used for this thesis and will not be discussed.

B. MOTION MONITORING AND CONTROL SYSTEM

Yamabico is a wheeled differential drive vehicle. Two separate 35 Watt DC motors drive each wheel via drive shafts using a 1:24 gear box ratio. The drive shafts are mounted with optical encoders to enable the robot to calculate how far each wheel has traveled in a given time step. This information is used to make odometry calculations. Four shock absorbing caster wheels are also mounted on the base of Yamabico to provide stability. Since this is an indoor vehicle, there was less emphasis on detailing physical factors such as slippage and uneven terrain effects and the more general approach of dead reckoning (DR) was used in movement calculations.

The DR approach does cause navigational problems over long running times. For instance, if the floor is extremely slippery and a high acceleration is used, the wheels may slip, resulting in a measured travel distance that is larger than the actual true distance of travel. As another example, if the robot drives over a board or an uneven floor segment, the robot may measure a turn when no turn was made. These errors are minor and acceptable over small distances. When the robot moves over much larger distances, the errors compound until eventually they are no longer negligible. At this point adjustments have to be made by the robot to realign itself with the real world. Several different

techniques can be used to update the robot's navigation. In a known or partially known world, a landmark can be used to correct navigational errors. Another technique, and the one used in this thesis, is to navigate by continuously updating its position and future path in reference to the world (i.e. follow a wall or walls.) This is explored in detail in Chapter IV.

The motion monitoring system interrupts the software system on regular 10 ms intervals. Yamabico's nominal speed is 30 cm/sec with a maximum speed of 65 cm/sec. At the nominal speed, interrupts occur every 0.3 cm as indicated by Equation (2.1).

$$30\frac{\text{cm}}{\text{sec}} \times 10^{-2}\ \frac{\text{sec}}{\text{interrupt}} = 0.3\frac{\text{cm}}{\text{interrupt}} \tag{2.1}$$

This short travel distance between motion updates provides a sufficiently high degree of accuracy in odometry calculations. During each interrupt, information from the drive shaft encoders is fed to a software interrupt routine. The software can then use this information to calculate its new position, make adjustments to future movement requirements, and return commanded linear and rotational velocities to the motion processor board. The board then translates these commanded velocities into electronic signals to the two wheel motors in a pulse-width modulation mode to effect the requested movement.

C. SONAR AND SONAR CONTROL SYSTEM

Yamabico's sonar system consists of twelve sonars arranged in a horizontal plane approximately 30 cm above ground level and angularly spaced in 30° increments to give full 360° coverage (see Figure 2). Each sonar is a transmitter/receiver pair operating at a

frequency of 40 kHz and may be individually enabled or disabled at any time within the user program.

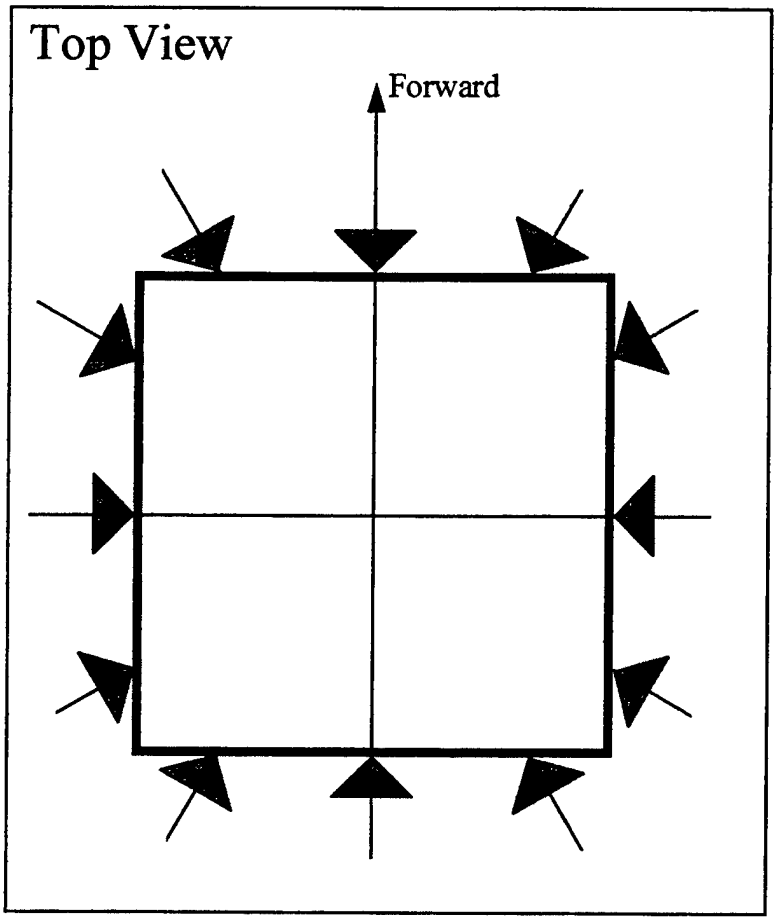


Figure 2. Sonar Configuration of Yamabico After Ref. [1].

The sonars are controlled by a separate processing board on Yamabico. In order to reduce sampling time, the sonars are organized into 3 groups of 4 sonars apiece. Each group contains sonars that are separated by 90° to minimize interference from neighboring sonar transmissions. All enabled sonars within the same group are pulsed simultaneously thereby reducing the time spent sampling by a factor of four when all twelve sonars are in use. The sonar groupings are shown in Table 1.

Group	Sonars in Group
0	0, 2, 5, 7
1	1, 3, 4, 6
2	8, 9, 10, 11

Table 1. Sonar Groupings.

As in the motion control hardware, the sonar system generates sonar interrupt events in the software. These interrupts occur at 50 ms intervals. Because the sonars are activated in groups, the time between pings of a single sonar can take up to 150 ms. When one or more sonars in a group are enabled, that group generates an interrupt and each enabled sonar in the group is activated. Similarly, if no sonars in a group are enabled, that group does not generate an interrupt. With these facts in mind, the time between pings on an single enabled sonar has a minimum value of 50 ms and a maximum value of 150 ms. This design is much more efficient than activating sonars individually, which could result in a maximum of 600 ms between pings.

In the original design of the sonar system, a pulse width of 1 ms was used. This was later changed to the current value of 0.5 ms to reduce the minimum range of the sonars. Assuming the speed of sound in air at room temperature is approximately 340 m/s, the minimum range can be calculated as half the distance the sound will travel in the time of a transmit pulse (pulse width).

$$\text{min range} = \frac{1}{2}\left(34000\frac{\text{cm}}{\text{sec}} \times 0.5\,\text{ms}\right) = 8.5\,\text{cm} \qquad (2.2)$$

The system actually sets the minimum range to be 9.6 cm to allow for circuit switching and settling times.

The maximum range of the sonars is imposed by hardware limitations. When a pulse is sent out, a 16 bit data register is used to count time until the pulse is received. Only the first 12 bits of this register are used for counting, limiting the counter to a maximum value of 4095 ($2^{12} - 1$). The counter is incremented every 6 μs which corresponds to an incremental distance of 0.204 cm. The maximum range of the sonar is thus:

$$\text{max range} = \frac{1}{2}(4095 \times 0.204 \text{ cm}) = 417.7 \text{ cm} \tag{2.3}$$

Again the system allows for circuitry timing and various other long range errors and sets the maximum range to be 409 cm.

The location of each sonar was depicted graphically in Figure 2. In order calculate global positions of the sonar returns, the software system maintains a table, similar to Table 2, of sonar positions relative to the center of the robot with the x-axis aligned to the front of the robot and angles measured counter-clockwise from the x-axis. Global positions are then calculated using the robot's current position, the relative position of the sonar making the measurement, and the distance returned by the sonar.

Sonar	X (cm)	Y (cm)	θ (rad)
0	23.6	- 0.5	0
1	-23.0	13.1	$5\pi/6$
2	-22.6	- 1.0	π
3	24.7	-14.6	$-\pi/6$
4	13.4	21.3	$\pi/3$
5	0.0	20.6	$\pi/2$
6	-12.6	-21.3	$-2\pi/3$
7	0.0	-20.5	$-\pi/2$
8	-13.4	21.3	$2\pi/3$
9	-23.5	-14.9	$-5\pi/6$
10	12.1	-21.3	$-\pi/3$
11	25.2	14.1	$\pi/6$

Table 2. Sonar Positions Relative to Center of Robot.

III. SOFTWARE LIBRARY

The software system used to control Yamabico is both complex and robust. Interrupts are used to optimize operations. Timing considerations of these interrupts are an important part of code development. The MML software library is composed of many files each containing software functions written in the C programming language. Multiple files are used to separate and organize the main objectives of each part of the system. These objectives can be roughly categorized as low level motion control, sonar control, and sonar analysis, and high level geometry, input/output, and user control.

A. INTERRUPT CONTROL SYSTEM

As mentioned in Chapter II, Yamabico operates in an interrupt based environment. The software library serves as a multi-tasking operating system for the robot [1]. It manages interrupt requests based on the relative importance of the task associated with each type of interrupt. When received, higher priority interrupts are handled immediately and lower priority tasks are suspended until completion of the corresponding interrupt handling routine.

The software system designates eight levels (0 - 7) of interrupt priorities. Table 3 depicts these levels with higher level numbers indicating higher priority. The interrupt type indicates how each interrupt level interacts with the rest of the system. Asynchronous interrupt events are processed immediately while synchronous events, as the name implies, are synchronized with circuitry timing.

Interrupt Level	Interrupt Source	Function	Interrupt Type
7	stop button	reset	asynchronous
6	--	not used	--
5	--	not used	--
4	Serial Board 1	locomotion	synchronous
3	Serial Board 2	teletype	asynchronous
2	Sonar Board	sonar	synchronous
1	Serial Board 0	debugger	synchronous
0	--	user instructions	none

Table 3. Interrupt Priority Table for Yamabico After Ref. [1].

A reset button is installed on top of Yamabico where it is readily available to the operator. When depressed, this button generates an interrupt of the highest level (level 7). It causes the robot to halt immediately and resets all hardware components. This button is used extensively during development of new routines for Yamabico because invariably these new routines have flaws in logic or unexpected side effects. By using the reset button, the operator may cease Yamabico's movement before a collision occurs.

Interrupt levels 5 and 6 are currently unused.

The motion system is incorporated at interrupt level 4. This is the highest priority interrupt that occurs during normal operation of Yamabico. Since motion hardware interrupts occur every 10 ms, this priority level reflects the importance of handling this

type of event immediately. All steering and odometry functions are run during the processing of this type interrupt.

All console operations of the laptop and data transfer between the laptop and the UNIX network have interrupt priority level 3. This allows for feedback to the outside user, input from the user, code uploading, and data downloading.

The sonar system operates at interrupt level 2 allowing sensor updating to occur at a relatively high priority. If no sonars are enabled, this level is bypassed during the interrupt polling procedure.

At the lowest levels are debugging routines at level 1 and the user control instructions at level 0. All other interrupts can override this last level. In general, level 0 can be thought of simply as code that runs whenever the system has time to run it. This last statement may seem misleading because the robot won't really do anything without the user code. It is in this part of the software that the initial state of the robot is set (i.e., position, initial speed, path, enabled sonars, data logging, etc.) and end of run conditions are checked. However, since these functions can be considered as initialization routines (before motion and sensing starts) or short status checks (during run), there are no adverse effects of placing them at the lowest interrupt level.

B. TRANSFORMATION MODEL OF MOTION

Before delving into the details of the motion algorithms in MML, an understanding of the underlying mathematical model must be gained. In most physical applications, Cartesian or polar coordinates are use to model kinematic effects. Dr.

Kanayama has developed a related but somewhat different and more versatile model of motion of a rigid body robot. This model is based on the fact that the trajectory of a rigid body object can be represented by a sequence of points, and since we calculate the vehicle's position at discrete intervals, the positional changes can be represented by a series of discrete transformations. These transformations are the heart of the motion model. This section presents a synopsis of Dr. Kanayama's transformation model of motion [2, 3, 4, 5, 6].

1. Transformations and Configurations

A transformation is described by three parameters, the conventional two-dimensional Cartesian coordinates (x, y) and an orientation angle (θ). In a rigid body vehicle, these correspond to a translation and a rotation (or heading change). Note that in this model, positive angles are measured in the counter-clockwise direction relative to a reference (local or global) x-axis. There is also the idea of a configuration. A configuration describes the current state of the vehicle using the same parameters of a transformation but with somewhat different meanings. In a configuration, x and y represent the global coordinates of a local reference frame and θ represents the current orientation of the local x-axis. Both transformations and configurations are specified by a triple (x, y, θ). The main difference between a transformation and a configuration is that the former acts as an operator and the latter acts as a spatial specification of an object. Figure 3 shows an example of an initial configuration (q_0) which undergoes a transformation (T_1) resulting in a new configuration (q_1).

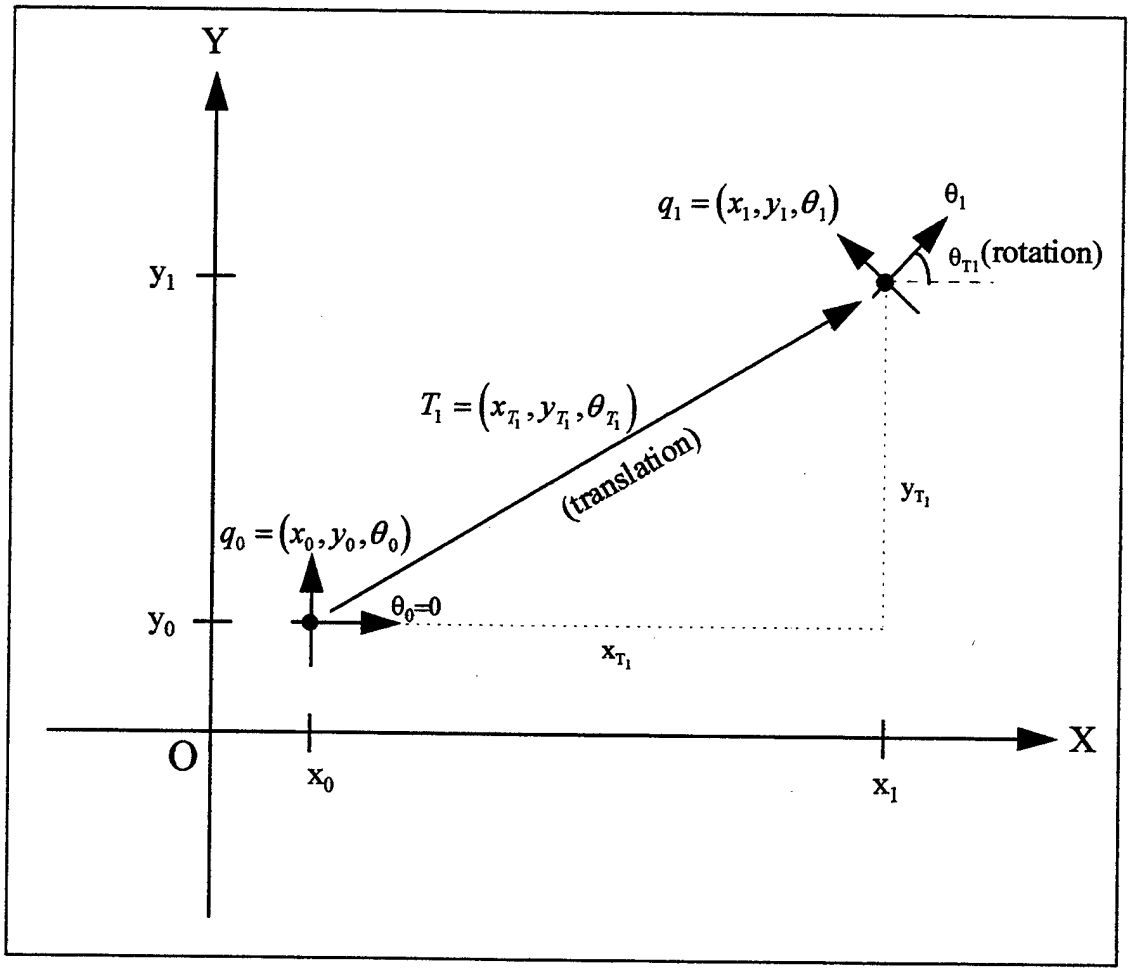


Figure 3. Example of Transformation of a Configuration.

2. Composition

In Figure 3, to get from q_0 to q_1, the transformation T_1 is applied to the configuration q_0 by a method called composition represented by the symbol (∘). Composition is defined as follows:

Let $q_0 = (x_0, y_0, \theta_0)$ *and* $T_1 = (x_1, y_1, \theta_1)$,
Then:

$$q_0 \circ T_1 = \begin{pmatrix} x_0 + x_1 \cos\theta_0 - y_1 \sin\theta_0 \\ y_0 + x_1 \sin\theta_0 + y_1 \cos\theta_0 \\ \theta_0 + \theta_1 \end{pmatrix} \tag{3.1}$$

In this thesis, the two forms seen above (i.e. parameter triple and column vector) will be used interchangeably to represent a transformation/configuration.

3. Inverse and Relative Transformation

In Figure 3, $q_1 = q_0 \circ T_1$. In order to get from q_1 back to q_0, an inverse operation is needed such that $q_1 \circ T_1^{-1} = q_0 \circ T_1 \circ T_1^{-1} = q_0$, where T_1^{-1} is the inverse transformation of T_1. When a transformation is composed with its inverse, it produces an identity transformation which neutralizes its effect in an expression such as that above. The definition of an inverse transformation is:

$$\begin{aligned} &Let\ T = (x, y, \theta), \\ &Then: \end{aligned}$$

$$T^{-1} = \begin{pmatrix} -x\cos\theta - y\sin\theta \\ x\sin\theta - y\cos\theta \\ -\theta \end{pmatrix} \tag{3.2}$$

One use for the inverse function is in finding relative transformations. Suppose a robot vehicle is represented by the global configuration q_V, and an object is represented by the global configuration q_O, as shown in Figure 4. To find the object's relative transformation from the vehicle, there must be a transformation q^* such that $q_V \circ q^* = q_O$. Composing both sides with the inverse of q_V results in $q^* = (q_V)^{-1} \circ q_O$, the transformation of the object relative to the vehicle's coordinate frame.

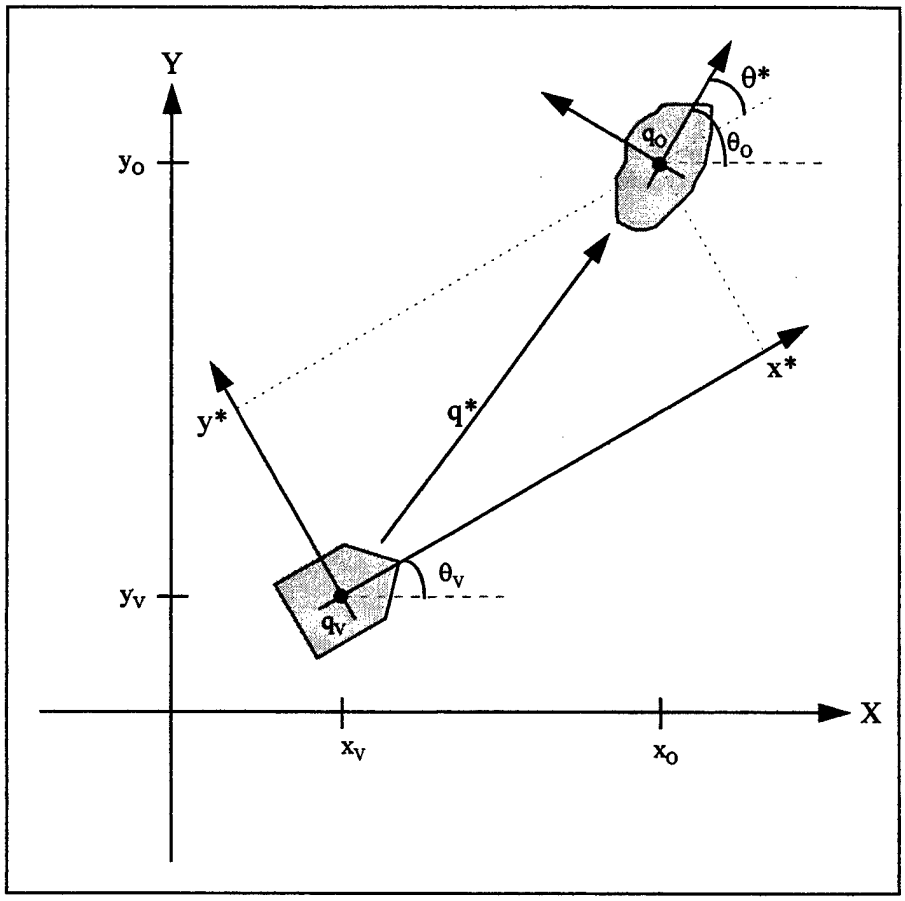


Figure 4. Example of a Relative Transformation (q*).

4. Lines, Circles, and Paths

For path planning, the transformation/configuration model is expanded so that lines and circular arcs can also be expressed as configurations. For this purpose, a configuration is redefined by adding a fourth parameter. This fourth parameter (κ) represents the curvature of the configuration where $1/\kappa$ is the radius of curvature. Lines become directed lines with the x and y parameters specifying its origin, θ its orientation, and since the radius of curvature of a line is infinite, $\kappa = 0$. For example, a line (L) beginning at the global position ($x = 1$, $y = 5$) and angled 45° above the x-axis would be

defined by the configuration $L=\left(1{,}5,\frac{\pi}{4},0\right)$. Circles are somewhat more complex, but follow similar reasoning. A circle configuration is a directed circle starting at the (x, y) position (on the circle), oriented in the θ direction (tangent to the circle), with curvature κ = 1/radius. For example, one representation of a counter-clockwise circle centered about the origin with radius r = 5 (κ = 1/r = 0.2) would be $C=\left(5{,}0,\frac{\pi}{2},0.2\right)$. Figure 5 depicts L and C.

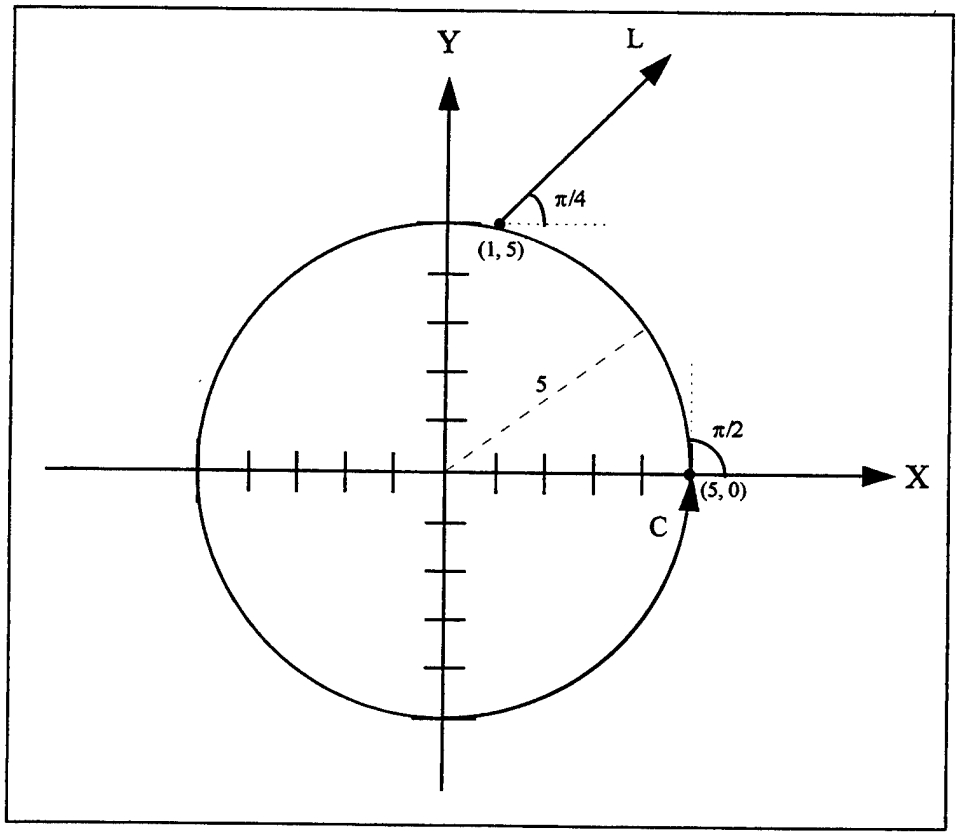


Figure 5. Example Line(L) and Circle(C) Configurations.

These ideas of transformations and configurations are a very powerful and can be applied in a variety of ways. For instance, to specify a simple square pattern starting at the origin, only one configuration and one transformation is needed. The following algorithm will produce four lines making the square shown in Figure 6:

1. $L_1 = q_0 = (0,0,0,0)$
2. $T = \left(100,0,\frac{\pi}{2}\right)$
3. **for** $n = 2$ **to** 4
4. $\quad L_n = L_{n-1} \circ T$

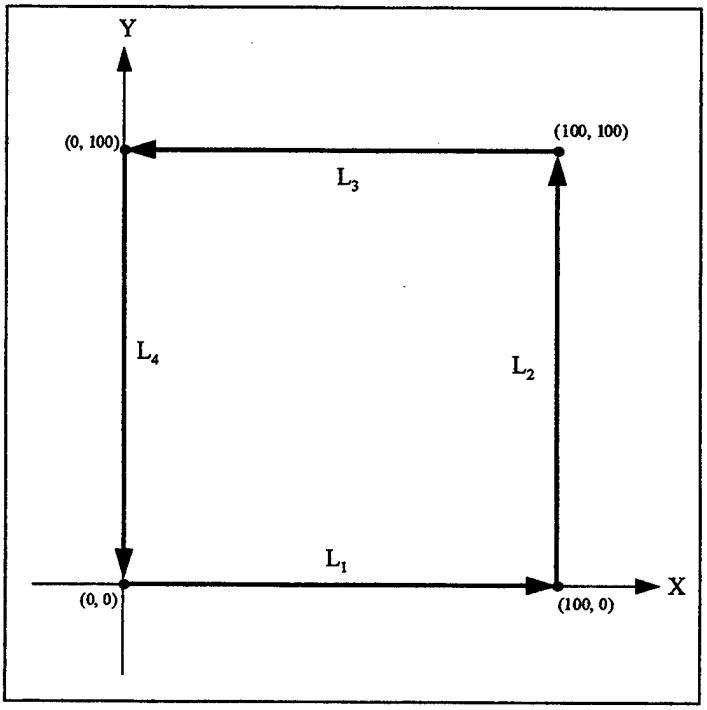


Figure 6. Square path pattern defined by an initial configuration and successively applying a transformation.

Path planning for Yamabico is accomplished in this manner, where a path is specified as a sequence of lines and circular arcs.

5. Circular Transformation

So far only large scale transformations have been depicted. Yamabico updates its odometry every 10 ms. In this small time period, the distance it moves is very small (~0.03 cm). Taking advantage of this fact another concept, called a circular transformation, is used in path trajectory calculation.

First, consider a path (P) which can be defined by an initial configuration $q_0 = (x_0, y_0, \theta_0, \kappa_0)$ and future configurations as a function of arc length s. The path is completely specified by κ as a function of s. Because the curvature is the derivative of the tangential orientation $\left(\kappa(s) = \frac{d\theta(s)}{ds}\right)$ along P, once κ(s) is defined, θ(s) is found by:

$$\theta(s) = \theta_0 + \int_0^s \kappa(u)du \tag{3.3}$$

Positions along P are also functions of s and are specified by the following parametric equations:

$$x(s) = x_0 + \int_0^s \cos(\theta(u))du \tag{3.4}$$

$$y(s) = y_0 + \int_0^s \sin(\theta(u))du \tag{3.5}$$

These equations are a continuous representation of a path, and the instantaneous configuration of a vehicle following this path is:

$$q(s) = (x(s), y(s), \theta(s), \kappa(s)) \tag{3.6}$$

In digital computer based systems, continuous functions are represented by a series of discrete changes or deltas (Δ). In Yamabico, a continuous path is represented by a series of discrete instantaneous configurations

$$q_0, q_1, q_2, \cdots, q_i, q_{i+1}, \cdots \tag{3.7}$$

Each successive configuration can be found by the following iterative technique:

$$q_{i+1} = q_i \circ \Delta q_i \tag{3.8}$$

where the Δq_i's are incremental transformations over a small path distance Δs.

If Δs is sufficiently small, we can use the following approximation to the amount of change in orientation over Δs:

$$\Delta\theta = \kappa(s)\Delta s \tag{3.9}$$

To improve this approximation we use the value of the curvature at the midpoint of the interval [s, s + Δs]:

$$\Delta\theta = \kappa\left(s + \frac{\Delta s}{2}\right)\Delta s \tag{3.10}$$

If it is also assumed that the curvature along the interval [s, s + Δs] remains constant, then the interval represents a circular arc of radius 1/κ. Figure 7 depicts this circular arc with curve length Δs and orientation change Δθ starting at the origin of a reference frame with initial orientation aligned with the x-axis.

The radius of the circular arc is:

$$r = \frac{1}{\kappa} = \frac{\Delta s}{\Delta\theta} \tag{3.11}$$

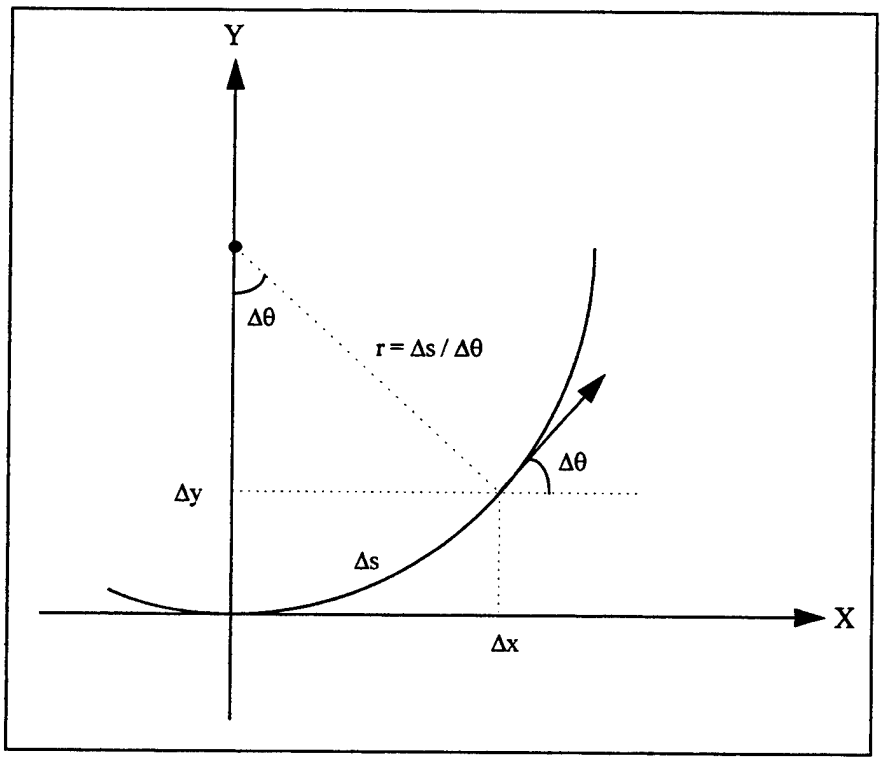


Figure 7. Circular Arc Parameters After Ref. [2].

An expression for Δq_i can now be formulated as a circular transformation, denoted by C(Δs, Δθ). First, for $\Delta\theta \neq 0$:

$$\begin{aligned} C(\Delta s, \Delta\theta) &\equiv (\Delta x, \Delta y, \Delta\theta) \\ &= (r\sin\Delta\theta, r(1-\cos\Delta\theta), \Delta\theta) \\ &= \left(\left(\frac{\sin\Delta\theta}{\Delta\theta}\right)\Delta s, \left(\frac{1-\cos\Delta\theta}{\Delta\theta}\right)\Delta s, \Delta\theta\right) \end{aligned} \tag{3.12}$$

For $\Delta\theta = 0$, the circular arc becomes a straight line and:

$$C(\Delta s, \Delta\theta) \equiv (\Delta s, 0, 0) \tag{3.13}$$

Finally, $|\Delta\theta|$ is normally small enough to justify using a Taylor expansion for the sine and cosine terms in Equation (3.12). With these expansions, Equations (3.12) and (3.13) can be condensed into a single equation:

$$C(\Delta s, \Delta\theta) = \left(\left(1 - \frac{\Delta\theta^2}{3!} + \frac{\Delta\theta^4}{5!} - \cdots\right)\Delta s, \left(\frac{1}{2!} - \frac{\Delta\theta^2}{4!} + \frac{\Delta\theta^4}{6!} - \cdots\right)\Delta\theta\Delta s, \Delta\theta\right) \quad (3.14)$$

Using Equation (3.14) any path can be generated using the following algorithm:

1. $s = 0$
2. $q = q_0$
3. **do forever**
4. $\Delta\theta = \kappa\left(s + \frac{\Delta s}{2}\right)\Delta s$
5. $\Delta q = C(\Delta s, \Delta\theta)$
6. $q = q \circ \Delta q$
7. $s = s + \Delta s$

This algorithm closely matches the algorithm Yamabico uses to calculate its trajectory while it is in motion. The robot must perform some extra steps such as calculating Δs by averaging the amount of rotation of each wheel and determining what kind of path shape it is following, but the algorithm presented above is central to computing the vehicle's current position.

6. Tracking Lines

In the preceding discussions a method for specifying a desired path for the robot (lines and circles) and a method for calculating the robot's trajectory (circular transformation) have been defined. In order to have the robot match its trajectory to the desired path, there must be a way to determine how far the robot is away from the path and how to proceed to the path. Just as the driver of a car uses the steering wheel to match the vehicle's trajectory to the desired path (the road), the robot uses a steering

function to accomplish this task. The first item to look at is using the steering function to track a line (see Figure 8).

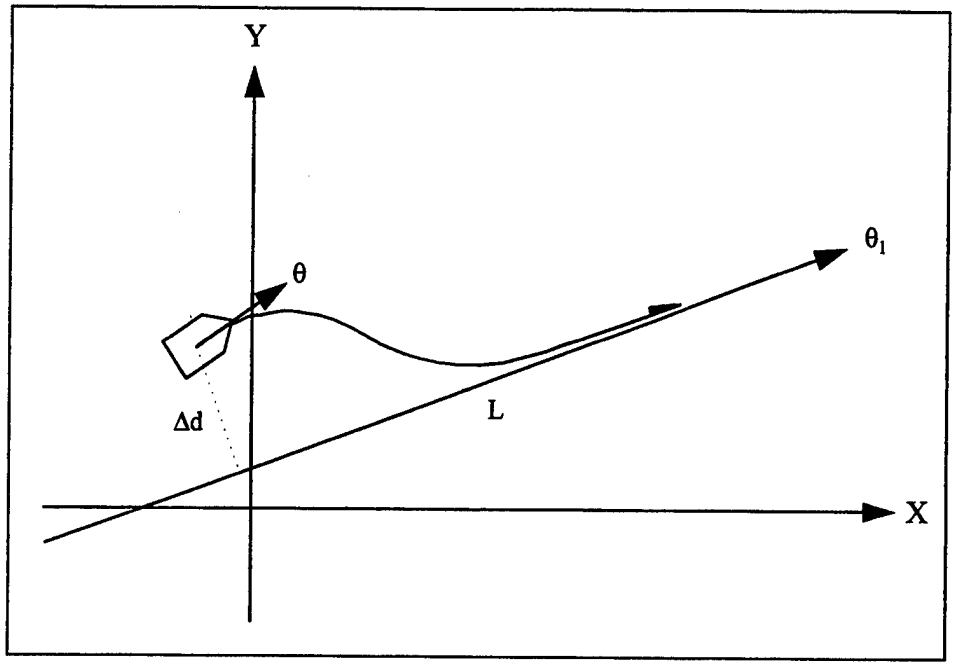


Figure 8. General Line Tracking After Ref. [2].

To generate a smooth trajectory, the curvature function must maintain continuity. To fulfill this requirement, the derivative of the trajectory curvature must remain finite at all times and is used as the definition of the steering function J(L):

$$\frac{d\kappa}{ds} = J(L) = -a\kappa - b\Phi(\theta - \theta_1) - c\Delta d \tag{3.15}$$

where a, b, and c are positive constants, Φ is an angular normalization function (normalizes angular range to the interval $[-\pi, \pi]$), θ_1 is the orientation of the line (L), and Δd is the signed distance from the line. For Δd, positive distances are to the left of the directed line and negative distances to the right.

The three terms on the right hand side of Equation (3.15) are negative feedback terms and the coefficients are feedback gain parameters. The first term -aκ is negative feedback for curvature error ($\kappa_1 = 0$). The second term -bΦ is negative feedback for the orientation error. The third term -cΔd is negative feedback for positional error (Δd is measured normal to L). When combined in this way, these terms allow the robot's trajectory to converge to the line (L) without overshoot or undershoot. However, this is only true for an optimal combination of the gain parameters (a, b, c) defined below:

$$\begin{aligned} a &= 3k \\ b &= 3k^2 \\ c &= k^3 \end{aligned} \tag{3.16}$$

where k is the gain of the steering function. See section 3.2 of Reference [5] for a detailed derivation of these parameters.

The smoothness of the line tracking motion is defined by:

$$\sigma = \frac{1}{k} \tag{3.17}$$

which represents a smoother motion with larger σ and a sharper motion with smaller σ. This parameter is the only parameter in the steering function J(L), and is used to control how fast the robot reacts to the feedback.

Figure 9 illustrates the effect of different σ values on the robot's trajectory when the robot starts at $q_0 = (0, 100, 0, 0)$ and tracks the x-axis, L = (0, 0, 0, 0).

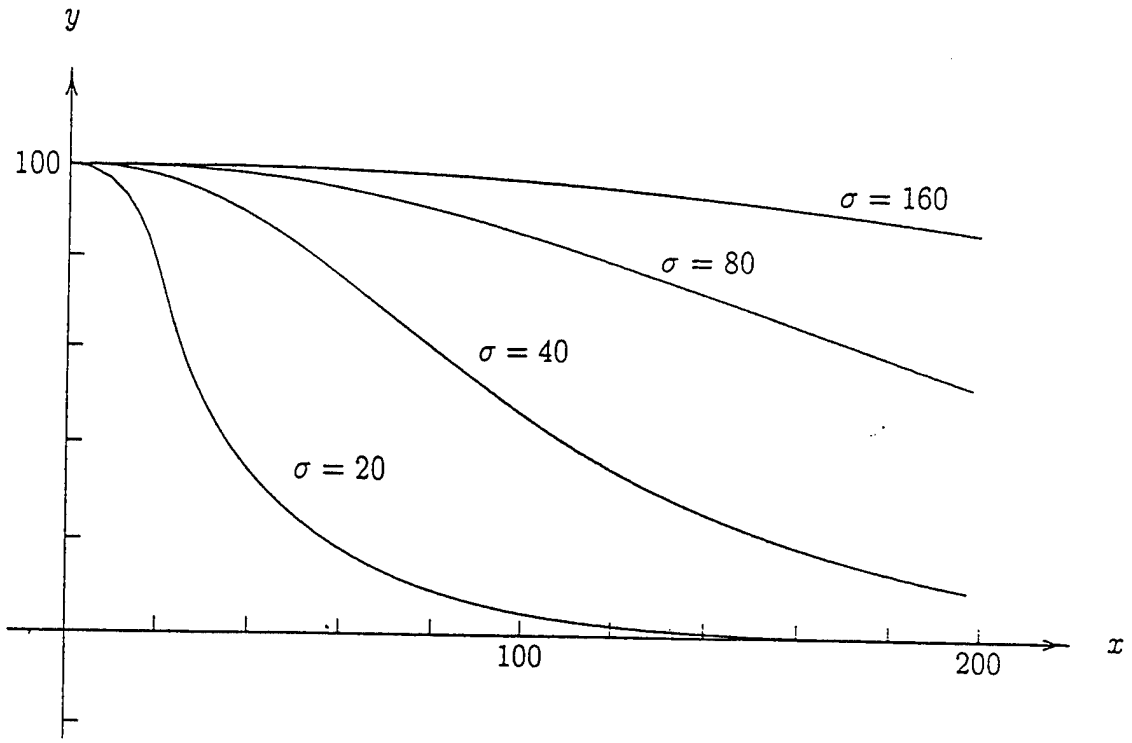


Figure 9. Effect of Smoothness Parameter From Ref. [2].

7. Tracking Circles

The steering function for directed circle tracking is similar to that of Equation (3.15). In line tracking, the curvature of the path (L) was zero. With circle tracking, the curvature of the path has a value other than zero and must be taken into consideration. This thesis does not make extensive use of circle tracking, and therefore the definitions of the steering function values are stated below without detailed explanation. Figure 10 depicts the parameters involved in circle tracking.

The steering function for tracking a circle (L) is:

$$\frac{d\kappa}{ds} = J(L) = -a(\kappa - \kappa_1) - b\Phi(\theta - \theta_1) - cd^* \qquad (3.18)$$

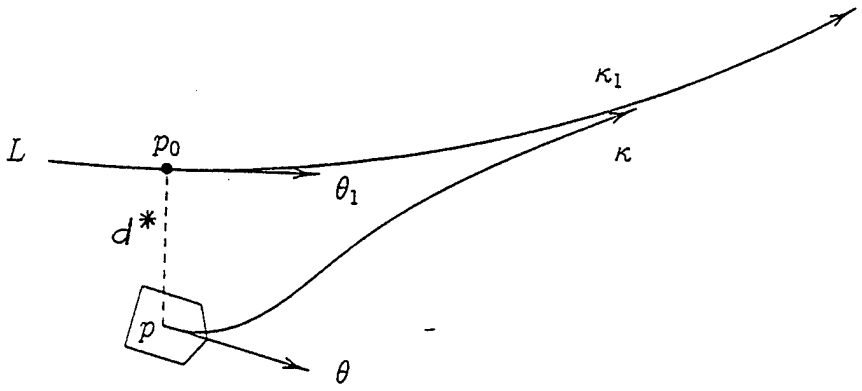


Figure 10. General Circle Tracking From Ref. [2].

where κ_1 is the curvature of the circle, θ_1 is the tangential orientation of the closest point on the circle, and d^* is the signed distance to the closest point on the. The gain coefficients are also effected by the curvature of L and become:

$$\begin{aligned} a &= 3k \\ b &= 3k^2 - \kappa_1^2 \\ c &= k^3 - 3k\kappa_1^2 \end{aligned} \tag{3.19}$$

References [2, 6] provide an in depth discussion on these topics.

8. Tracking Paths

The steering function presented in the previous two sections tells us how to track a single directed line or directed circle. A path can be defined by a sequence of path segments (lines and circles), such as the square path in Figure 6. When changing between the current path segment and the next sequential path segment, a neutral switching point is defined to prevent unnecessary early turns or overshoots. This is accomplished by maintaining two steering function values, $J(L_i)$ for the current path segment and $J(L_{i+1})$

for the next path segment. The neutral switching point occurs when these two values are equal, $J(L_i) = J(L_{i+1})$. Since these steering values are floating point numbers, the check is made in program code by monitoring the sign of $J(L_i) - J(L_{i+1})$. When the sign changes, the neutral switching point has been reached. This approach avoids checking the equivalence of two floating point numbers, which can lead to errors. The algorithm shown below demonstrates how neutral switching is implemented in Yamabico when following a defined path P:

1. $q = q_0$
2. $P = (L_1, L_2, \ldots, L_n),\ n \geq 1$
3. **do forever**
4. **if** $\left(n \geq 2 \text{ and } \left(\text{the sign of } J(L_1) - J(L_2) \text{ has changed}\right)\right)$
5. **then** shift pathsegments $(L_1 \leftarrow L_2, L_2 \leftarrow L_3, \ldots)$
6. $\kappa = \kappa + J(L_1)\Delta s$
7. $q = q \circ C(\Delta s, \kappa \Delta s)$

C. THESIS RELATED MML ROUTINES

The previous section (Section B) showed how the transformation model of motion can be used for path planning, trajectory calculation, and relative positioning computation of objects in the world of a rigid body robot. This section discusses how this model is used in MML to direct the motion of Yamabico.

1. Motion Modes

Yamabico is primarily controlled by two types of motion modes. The first of these is a track mode and the second a rotate mode. The track mode is further broken down into two types, the normal track mode in which lines and circles are tracked in

sequence, and the track stop mode where the current line or circle is tracked while the robot comes to a halt. The rotate mode is simply a turn in place type of motion.

By using these modes, many types of paths can be constructed for and followed by Yamabico. The user specifies the path by issuing a sequence of track and/or rotate commands in the user control file. For each track or rotate command, a corresponding path element is inserted into an instruction buffer (or queue) to be extracted by the motion control routines. Figure 11 presents a block diagram of this process.

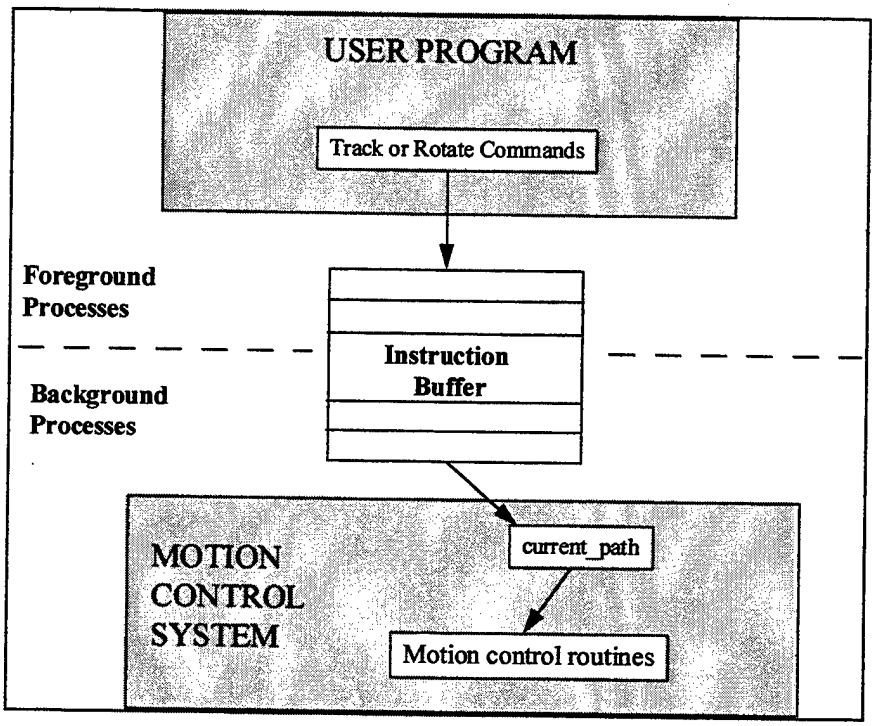


Figure 11. Motion Control Setup After Ref. [1].

The process begins in the file **<user.c>**, included in Appendix B. The user defines a sequence of configurations (x, y, θ, κ) and issues a track (function *track( )*) or a rotate (*Rotate( )*) command for each configuration. The final configuration in the sequence is issued with a track stop (*trackS ( )*) command so that motion will stop when

the sequence is complete. The *track( )/Rotate( )/trackS( )* command functions are located in the file **<seqcmd.c>** (Appendix A). They take each configuration and add path type information, steering related information, and circle information if the configuration specifies a circle. The resulting path element is stored in a data structure called a LINE, defined in **<definitions.h>** (Appendix A). This LINE is then inserted into the instruction buffer via the function *AddLine( )*, located in **<queue.c>** (Appendix A).

This process does not directly control Yamabico's motion. In other words, the sequence of configurations is a list of directions for Yamabico to follow in the future. The motion control routines in **<motion.c>** (Appendix A) are the ones that control Yamabico. They receive the instructions from the buffer and carry them out according to an algorithm like the one mentioned in Section B.8. The algorithm is actually spread out over several functions and files. Lines 1 and 2 of that algorithm are executed by the user program. The initial configuration is either specified by the user or a default value is used. The path P is specified by the instruction buffer built by the track and rotate commands and the current path element is kept in the static variable *currentPath* local to the file **<motion.c>**.

The loop in lines 3-7 of the algorithm is executed once every motion control cycle (10 ms). When the system signals a motion interrupt, the motion interrupt handler is called. This function, *MotionSysControl()* in **<motion.c>**, is the workhorse of the motion control system. It first updates Yamabico's movement by calculating how far each wheel has traveled during the last motion control cycle. It then uses this information to compute its path change (Δs) and orientation change ($\Delta\theta$).

After updating a variable keeping track of the total distance the robot has traveled the *MotionSysControl()* function calls the *localize()* function which uses the calculated Δs and $\Delta\theta$ to update the vehicle's current configuration by circular transformation. The localization corresponds to line 7 of the algorithm. According to the algorithm, this step is out of order. This is due to the fact that Δs and $\Delta\theta$ are computed directly from the hardware and can be used immediately. The ordering difference is negligible since κ changes only a small amount during the motion control cycle.

The next step in *MotionSysControl()* calculates the actual linear and rotational velocities maintained by the robot. These are used in later calculations. At this point all motion data from the last motion control cycle has been calculated and *LogMotion()* is called to log the motion data if logging has been turned on.

MotionSysControl() next calls *motionRules()* to determine the commanded linear and rotational velocities for the next motion control cycle. The *motionRules()* function performs all the calculations involving future movements. It encapsulates the functionality of lines 4-6 in the algorithm. Depending on the motion mode (track or rotate), it directs program execution to an appropriate motion rule handling function. The motion rule function (*trackRule()* or *rotateRule()*) is responsible for performing steering function calculations, extracting the next path element from the buffer if appropriate, updating κ, and returning the commanded velocities for the next motion control cycle.

The returned commanded velocities are relayed back through the *motionRules()* function to the *MotionSysControl()* function where they are translated into commanded

velocities for each wheel by the function *SetMovement( )*. Once this is done, the system clock is updated and an external LED is blinked for visual feedback. This completes one motion control interrupt cycle and control is returned back to the main system where other computations are performed or other interrupts are handled.

2. Linear Fitting

Each sonar on Yamabico is capable of gathering and maintaining collections of data in both raw form (distance from sonar) and in global form (global position of the return). The related data structures are contained in the file **<sonar.h>** (Appendix A) and related functions in **<sonarmath.c>** (Appendix A) . Some of these functions provide the ability to fit a line to a collection of global positions using the least-squares summation technique.

a) Equal Weighted Points

The following discussion of least-squares linear regression analysis is extracted from Ref. [1] and follows techniques covered in most probability and statistics textbooks, such as Ref. [7].

For a set of equally weighted unique points, a line that best fits the points can be defined as that line which minimizes the sum of the squares of the distances from the points to the line. The distance of a point to the line is called the residual and is denoted by δ. The goal is therefore to minimize the sum $S = \sum \delta^2$.

First, let $R = \{p_1, \ldots, p_n\}$, where $n \geq 2$ and $p_i = (x_i, y_i)$, be the set of n points to be fitted. The moments m_{jk} $(0 \leq j + k \leq 2)$ of R are defined as:

$$m_{00} = \sum_{i=1}^{n} 1 = n \tag{3.20}$$

$$m_{10} = \sum_{i=1}^{n} x_i \tag{3.21}$$

$$m_{01} = \sum_{i=1}^{n} y_i \tag{3.22}$$

$$m_{11} = \sum_{i=1}^{n} x_i y_i \tag{3.23}$$

$$m_{20} = \sum_{i=1}^{n} x_i^2 \tag{3.24}$$

$$m_{02} = \sum_{i=1}^{n} y_i^2 \tag{3.25}$$

The centroid C of the set R is given by:

$$C = (\mu_x, \mu_y) = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \tag{3.26}$$

The secondary moments about C are:

$$M_{11} = \sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y) = m_{11} - \frac{m_{10} m_{01}}{m_{00}} \tag{3.27}$$

$$M_{20} = \sum_{i=1}^{n} (x_i - \mu_x)^2 = m_{20} - \frac{m_{10}^2}{m_{00}} \tag{3.28}$$

$$M_{02} = \sum_{i=1}^{n} (y_i - \mu_y)^2 = m_{02} - \frac{m_{01}^2}{m_{00}} \tag{3.29}$$

In finding a line L to represent a best fit, a parametric representation of L is adopted using the line's normal direction α and distance r from the origin O. This

approach is different from standard linear regression analysis but has the advantage of not having any singularities with respect to vertical lines. The line is therefore defined in terms of these parameters, L = (r, α) (see Figure 12).

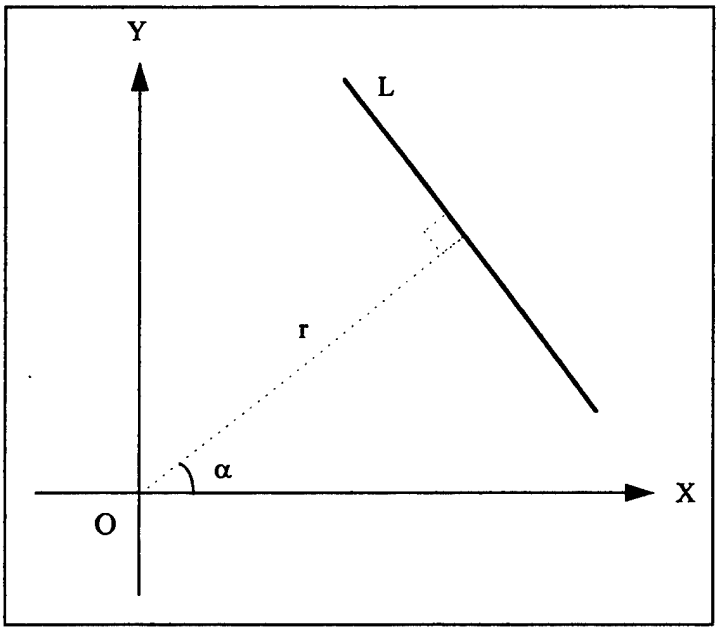


Figure 12. Parametric Representation of Linear Fitted Line After Ref. [2].

Note that every point (x, y) on the line L = (r, α) satisfies the relation:

$$x\cos\alpha + y\sin\alpha = r \tag{3.30}$$

which implies that the residual of the point is:

$$\delta = x\cos\alpha + y\sin\alpha - r \tag{3.31}$$

Also note that r is negative if the projection of O onto L is in the second or third quadrant (Figure 13).

Finally, the sum of squares of the residuals becomes:

$$S = \sum_{i=1}^{n} \delta^2 = \sum_{i=1}^{n} \left(\left(x_i \cos\alpha + y_i \sin\alpha \right) - r \right)^2 \tag{3.32}$$

In order to minimize the expression in Equation (3.32) the following condition must hold:

$$\frac{\partial S}{\partial r} = \frac{\partial S}{\partial \alpha} = 0 \tag{3.33}$$

Solving Equation (3.33) yields the following expressions for r and α:

$$r = \mu_x \cos\alpha + \mu_y \sin\alpha \tag{3.34}$$

$$\alpha = \frac{1}{2}\text{atan2}\left(-2M_{11}, M_{02} - M_{20}\right) \tag{3.35}$$

where *atan2()* is a more robust arctangent function found in most computer language libraries. It eliminates quadrant ambiguities returning an angle in the range $[-\pi, \pi]$ and therefore, by Equation (3.35), α is restricted to the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$. The four possible quadrant outcomes for L are depicted Figure 13.

The linear fitting routines in Yamabico use this procedure to calculate L for the global positions of sonar returns. Each of the twelve sonars is able to do this independently and stores current linear fitted data in their own data structure. When a new sonar return is received, the point data is used to update the linear fitted data by adding it to the moment summations in Equations (3.20-3.25). For instance, let (x_{n+1}, y_{n+1}) be the new point and m'_{jk} $(0 \le j + k \le 2)$ be the current value of each moment. The new moment summations m_{jk} would then be calculated as follows:

$$m_{00} = m'_{00} + 1 = n + 1 \tag{3.36}$$

$$m_{10} = m'_{10} + x_{n+1} \tag{3.37}$$

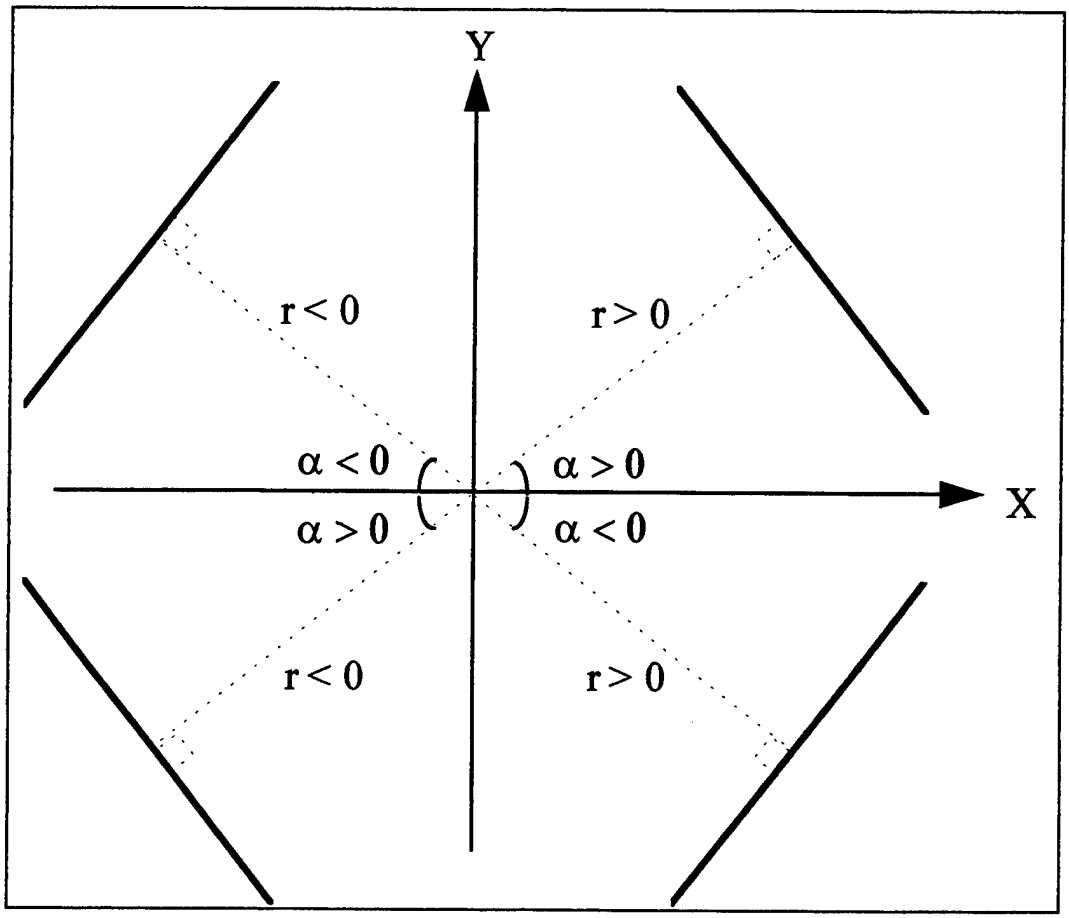


Figure 13. Four Quadrant Possibilities for Linear Fitted Line.

$$m_{01} = m'_{01} + y_{n+1} \tag{3.38}$$

$$m_{11} = m'_{11} + x_{n+1}y_{n+1} \tag{3.39}$$

$$m_{20} = m'_{20} + x^2_{n+1} \tag{3.40}$$

$$m_{02} = m'_{02} + y^2_{n+1} \tag{3.41}$$

The centroid and secondary moments are then recalculated using these new values, followed by a recalculation of r and α yielding a new line L. When the distance from a new point's location to the current L exceeds a threshold value,

processing of the current line segment stops, its endpoints are stored, the moments are reset, and calculation of a new line segment is begun using the new point data as the first point.

This process is repeated on every sonar control cycle for each sonar enabled for linear fitting. The building of line segments can be used to map a room, determine the shapes of objects, or update navigation by comparing the line segments to a map. All of these uses are passive and, except for the last one, do not contribute to the robot's motion analysis.

b) Using a Decay Factor

In order to follow a wall with sonar, a more reactive technique must be applied. Through conversations with Dr. Kanayama a modified version of the standard least-squares algorithm was developed for use in Yamabico. This modified version uses the idea of a decay factor applied to current linear data when adding a new point.

In Equations (3.36-3.41) there is an implicit multiplicative factor of one applied to the m'_{jk} terms. This follows from the stipulation that all points are equally weighted. If this factor is changed to a value less than one, then as new points are added, the effect of previous points on the linear data would decay in significance. In other words, there is still an overall history of previous data but new points have a greater impact on the calculation of the line L. In effect, the new point has a weight of one, the previous point has a weight of DF (where DF = decay factor), the point prior to that has a weight of $(DF)^2$, etc. Rewriting Equations (3.36-3.41) to reflect this change results in:

$$m_{00} = DF \times m'_{00} + 1 = DF \times n + 1 \tag{3.42}$$

$$m_{10} = DF \times m'_{10} + x_{n+1} \tag{3.43}$$

$$m_{01} = DF \times m'_{01} + y_{n+1} \tag{3.44}$$

$$m_{11} = DF \times m'_{11} + x_{n+1} y_{n+1} \tag{3.45}$$

$$m_{20} = DF \times m'_{20} + x^2_{n+1} \tag{3.46}$$

$$m_{02} = DF \times m'_{02} + y^2_{n+1} \tag{3.47}$$

Each time a new point is added, the moments are recalculated in this manner and, as in the equal weighted procedure, the centroid, secondary moments, r, and α are recalculated to give a new representation for the line L. This technique yields good response to a changing environment. It is reactive in that instead of having to start a new line segment if a new point is beyond a certain threshold, as mentioned in the previous section, the line L is adjusted to fit the new data. In this way, the line constantly changes and adjusts to the environment.

Figure 14 shows the effect of three different decay factors (< 1) compared to no decay factor (= 1) on the linear fitted line. It simulates a robot traversing a wall and taking sonar readings. The wall is represented by the connected solid line. The global position of the sonar return from the wall is represented by the intersection of the dashed vertical lines and the wall. The orientation of the linear fitted line is depicted by the solid line segments at the top of the dashed lines.

Analysis of Figure 14 results in the conclusion that smaller decay factors provide greater response to new data. For wall following, the robot should react to

changes in a timely manner while maintaining a smooth and continuous path curvature. Decay factors that are too large (i.e. close to one) violate the first stipulation while decay factors that are too small violate the second stipulation. It was decided through empirical observation that a decay factor of 0.9 would be used in the linear fitting with decay factor algorithms.

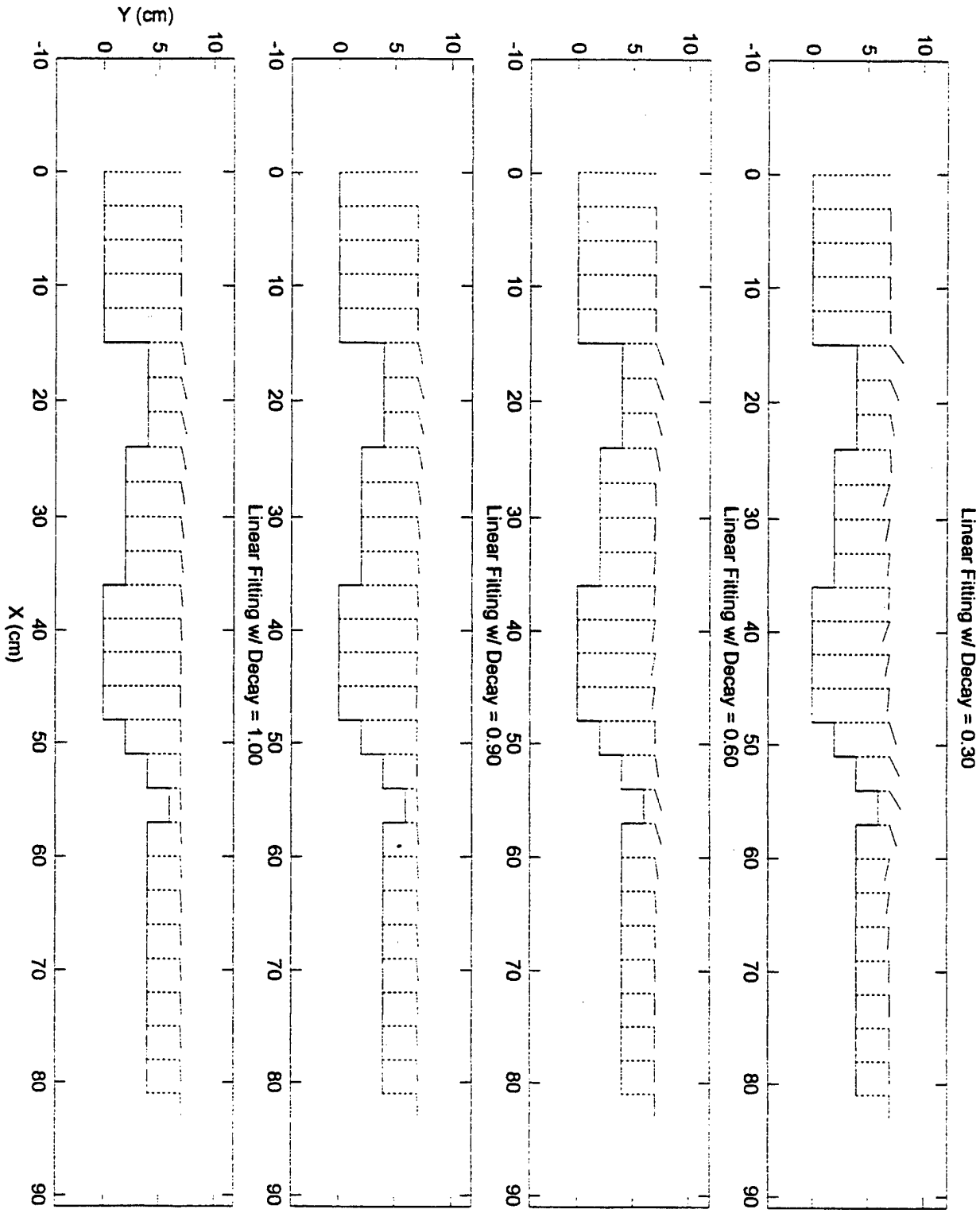


Figure 14. Linear Fitting Decay Factor Effect.

IV. WALL FOLLOWING AND OBSTACLE AVOIDANCE

The foundation for solving the problem defined for this thesis lies in having Yamabico maintain an offset from a wall while moving in the forward direction. The first attempt in realizing this goal was to call existing MML functions from the user file to direct Yamabico's motion. This approach proved to be flawed and it was determined that implementing a new type of motion mode would be the best method for solving the problem. Implementation of this new motion mode required modifications to the MML software library.

This chapter discusses why the first solution attempt failed, how a template of a new motion mode was added to MML, and how this template was filled in to solve the problem of following a wall.

A. FIRST ATTEMPT

Section C.1 of Chapter III explained how currently implemented motion modes are used to define a path that Yamabico will follow. For instance, if the user wanted the robot to follow a square path, such as the one described by Figure 6, the following code would be created in the **<user.c>** file:

```
CONFIGURATION q0, L, T;
int n;

L = q0 = defineConfig(0, 0, 0, 0);
T  = defineConfig(100, 0, PI/2, 0);
setRobotConfigImm(q0);

for (n = 1; n <= 4; i++) {
   track(L);
   L = compose(&L, &T);
}

L = defineConfig(150, 0, 0, 0);
trackS(L);
```

In this example, *CONFIGURATION* is a data structure defined in **<definitions.h>** holding configuration parameters (x, y, θ, κ), *setRobotConfigImm()* is a function defined in **<motion.c>** which immediately sets the robot's configuration, *compose()* is a function defined in **<geometry.c>** (Appendix A) implementing the composition (∘) operation, and *track()/trackS()* are the functions discussed in Chapter III. The *for* loop creates the square path by successively calling the *track()* function with the current line configuration L, which adds L to the instruction buffer, and then composes L with the transformation T to get the next line. The final two lines of code create the last line configuration in the sequence so that the robot will stop when the sequence is complete.

This procedure works well for predefined paths because it sets up the instruction buffer before any motions, and hence motion interrupts, occur. There is another function defined in the file **<queue.c>** called *FlushBuffer()*, which flushes the instruction buffer. This function is a low level function, similar to *AddLine()*.

The first attempt at a solution for wall following tried to use these low level functions, *FlushBuffer()* and *AddLine()*, in the high level user program. The idea was to track an initial line until sufficient linear fitted data on sonar returns from the wall was acquired. Once the robot had good wall data, it would construct a line configuration from the linear data, flush the buffer, and track the new line by adding it to the instruction buffer. This solution idea was flawed because of timing differences between the motion interrupt cycle, the sonar interrupt cycle, and the user program. The low level instruction buffer routines were not intended to be used in the high level user routines. This fact was not originally known to the author, but was realized after careful analysis of the code.

The problem was that, once robot movement commenced, the motion interrupt routine had a higher priority than the sonar interrupt routine which, in turn, had higher priority than the user program (Table 3). The only possible way to fix this problem was to disable interrupts (normally bad idea in interrupt driven programs) during critical sections of code in the user program. However, this still caused problems. If a disable interrupt command was given in the user code, there was no guarantee that the command would be carried out immediately because this command was contained in a low priority section of the code. These timing problems were too much to overcome for solving a quick reaction type of problem and this first solution attempt was discarded.

B. IMPLEMENTING A NEW MOTION MODE

To implement a reactive type of motion mode in MML, the use of an instruction buffer must be bypassed. The functionality of this new motion mode should closely

mirror that of the tracking motion mode, but should also be self-sufficient without holding path elements in a buffer. In other words, the user program should be able to call a routine which initializes the system to perform the tasks of the new motion mode, but the tasks should be performed by the lower level functions.

In the tracking motion mode, the user calls *track()/trackS()* (in **<seqcmd.c>**) to initialize the system by building the instruction buffer. After the initialization is complete, the motion interrupt handling routine, *MotionSysControl()* in **<motion.c>**, calculates commanded velocities for the robot on each interrupt cycle by calling *motionRules()*. This function determines the current type of motion mode (e.g. *TRACKMODE*) and calls the appropriate motion rule function, such as *trackRule()*. The motion rule function handles the details of steering (*steer()*), determining the next path element via instruction buffer management (*ReadNextLine(), GetLine()*), and current path updating. After computations are complete, commanded velocities are returned to the motion interrupt handling routine and are applied to the robot.

The new motion mode should behave in a similar manner. The user should be able to call a function *trackXXX()*, which initializes the system for the new mode (XXX). This function should be contained in a file like **<XXXcmd.c>**. After initialization, the motion interrupt handling routine would call *motionRules()*, as before. This function should then choose an appropriate motion rule, *trackXXXRule()*, based on the current motion type, *TRACKXXXMODE*. The *trackXXXRule()* function should handle the specific steering computations (*steerXXX()*), update the current path, and return commanded velocities. Note that since this will be a reactive motion mode, there won't

be a next path element, only adjustments to the current path. Figure 15 shows a block diagram of this process.

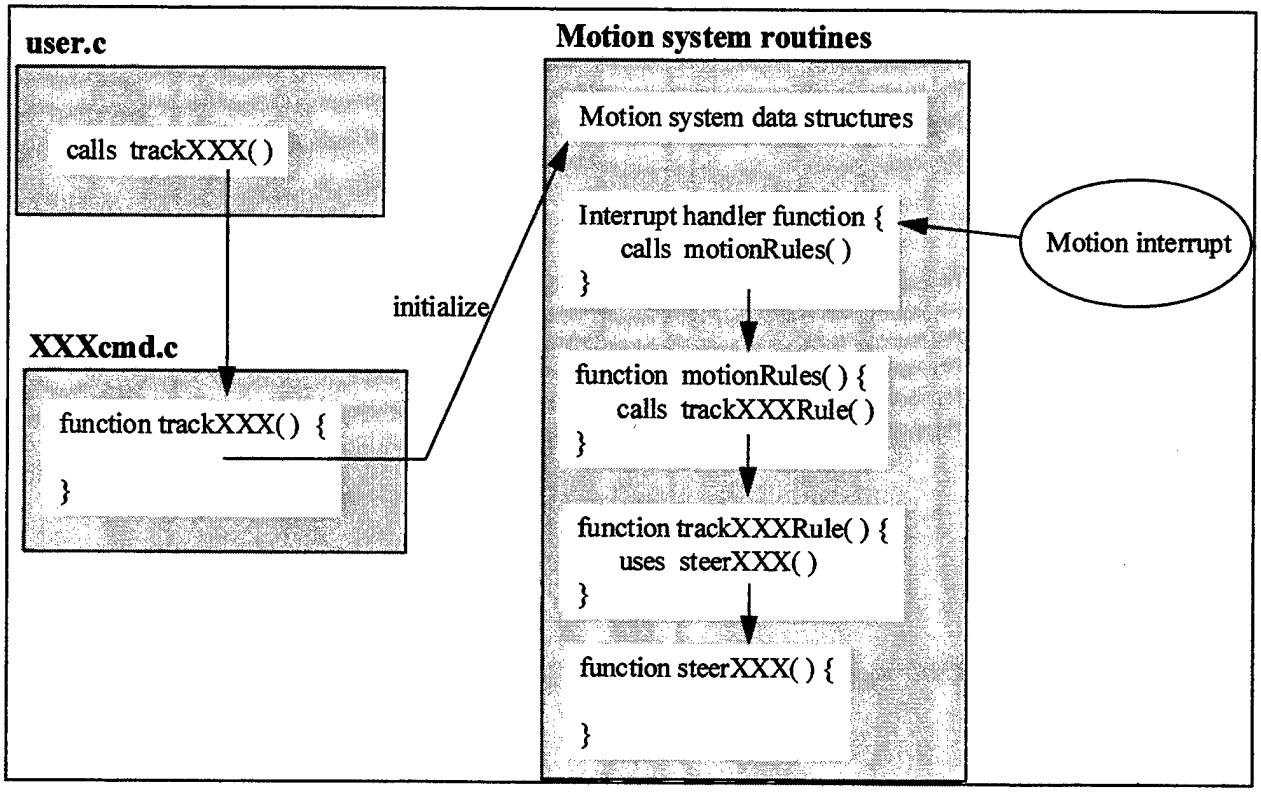


Figure 15. New Motion Mode Process.

The following sections discuss how these template forms (XXX) were implemented to support the wall following motion mode.

1. The User Program Function Call

At the highest level, the user program initializes the system for wall following by calling the function *trackWall(sonarNum, offset)*. This function is used to specify whether wall following will occur with the wall on the left side or the right side of the robot. It is also used to specify what the offset distance will be from the wall to the robot.

In the wall following mode, the robot uses linear fitted data from the wall as its navigation source. The *trackWall( )* function is used to set up the sonar system to perform this task. The user is able to specify which sonar to use by passing the sonar number in through the first argument, *sonarNum*. This has the added effect of specifying which side the wall will be on (#7 for right side, #5 for left side).

Constants are used in the code instead of sonar numbers to improve readability. These constants are defined in the file **<sonar.h>** and have the form *S###*, where the *S* signifies "sonar" and ### is a three digit number specifying the angle of the sonar measured clockwise relative to the front of the robot. For example, the right hand sonar (#7) is defined as *S090*, the left hand sonar (#5) is *S270*, and the front sonar (#0) is *S000*.

The second argument to *trackWall( )* is *offset* and is used to specify the offset distance the robot should use when following the wall. This distance is the magnitude of the offset measured from the center of the robot to the current linear fitted data representing the wall.

2. The *trackWall( )* Function

The *trackWall( )* function is located in the new MML file **<wallcmd.c>** (Appendix B). The function prepares the motion system and the sonar system for wall following.

For the motion system setup, it creates and sets field values in a *LINE* variable called *pathElement*. *LINE* is a data structure containing fields for the motion mode type, a configuration, steering function gain parameters, circular arc parameters, and other motion variables and is defined in **<definitions.h>**.

Depending on the sonar number argument, *trackWall( )* first assigns one of two values to the *Type* field: *TRACKRWALLMODE* for following a wall to the right of the robot, or *TRACKLWALLMODE* for following a wall to the left of the robot. These values are enumeration values of *MODE*, which is defined in **<definitions.h>**. The offset value is also set at this point depending on the sonar number. If the sonar number is *S090* then the wall will be on the right side. This implies that the robot will be to the left of the directed line representing the wall. Recalling the discussion in Chapter III (Section B.6), positive distances are to the left of a directed line and negative distances are to the right. When the sonar number is *S090*, the offset will be positive because the robot should maintain the offset distance to the left of the wall. On the other hand, when the sonar number is *S270*, the offset will be negative because the robot should maintain the offset distance to the right of the wall. The user specified offset is also checked to make sure its value will not cause the edge of the robot to come too close to the wall. If it will, then a default value is assigned.

trackWall( ) next initializes *pathElement* 's configuration to a default value (all zeroes) and its steering gain parameters using the robot's σ value. The initial path will be a straight line, so the circle parameters of *pathElement* won't initially be used by the motion system. To be safe, the center point is set to zeroes. The radius, however, is used to pass the offset value into the motion system. Once *pathElement* has been set up, it is used as an argument to the function *setPathElement( )*. This function is defined in **<motion.c>** and simply copies its argument into *currentPath*, a static file scope variable

used by many of the motion functions. This simple functionality is important because it is the only way for a routine outside of **<motion.c>** to set *currentPath.*

After *pathElement* and *currentPath* have been set, *trackWall( )* matches Yamabico's initial configuration to the initial path segment by calling *setRobotConfigImm( )* with *pathElement*'s configuration as the parameter. This function is located in **<motion.c>** and immediately sets the robot's configuration to the specified argument.

Finally, *trackWall( )* sets up the sonar system for wall tracking by enabling the required sonars (*S000* for both left and right wall following and either *S090* or *S270* depending on the sonar number argument) and enabling linear fitting using a decay factor. The functions used to do this, *EnableSonar( )* and *EnableLinearFittingDecay( )*, are located in files **<sonarcard.c>** (Appendix A) and **<sonarmath.c>** respectively. *EnableSonar( )* is a hardware related function. It takes a sonar number as an argument and adds that sonar to an internally stored enabled sonar list. *EnableLinearFittingDecay()* sets up the specified sonar to perform the linear fitting with decay computations discussed in Chapter III (Section C.2.b).

At this point, all initialization is complete, control is returned to the user program, and the motion and sonar control systems begin handling movement and sonar events.

3. The *wallHugRule( )* and *steerByWall( )* Functions

Once motion has started, the motion interrupt handling routine will call the *motionRules( )* function to obtain commanded velocities for the next motion cycle. In the wall following mode, this function will in turn call the new *wallHugRule( )* function, in

<motion.c>, which updates the current path element and calculates the commanded velocities to keep the robot on the current path. The *wallHugRule()* function is essentially the heart of the wall following motion mode. It uses both the currently implemented *steer()* function and a new *steerByWall()* function to calculate commanded velocities. The *steerByWall()* function calculates steering function values based on the wall represented by the decay version of linear fitting. Both of these new functions form the foundation for solving the problem presented in this thesis and are explained in more detail in the following sections.

C. FOLLOWING A STRAIGHT WALL

In order to follow a wall using sonar, the first thing that must be accomplished is finding a way to represent the wall based on the sonar returns. Once this representation is found, the next step is to find a way to direct the robot's motion so that it maintains a defined offset distance from the wall.

1. Method for Calculating Linear Representation of Sonar Returns

Once a sonar has been enabled to perform linear fitting calculations, it calculates and maintains the linear data in a structure called *SEGMENT_RES_DECAY* which is defined in **<sonar.h>**. This structure keeps all the important data for the linear fitting with decay calculations such as the number of points in the data set, the moments of the set, the decay factor, and the parameters r and α that define the linear fitted line. It also has two helper fields, one containing the sonar number and the other a flag indicating whether the linear data is usable or not.

There is one other important sonar data structure called *SONARD*, also defined in **<sonar.h>**. It contains all necessary sonar data for a particular sonar including boolean fields indicating whether normal and/or decay linear fitting is turned on, and if the sonar data has been updated recently. Other fields hold data related to the current sonar return including the global position, current and previous range, and the robot's configuration (position and orientation) at the time of the reading. There are also fields containing the sonar's configuration relative to the center of the robot. The sonar system maintains an external table of these *SONARD* data structures, one for each sonar and indexed by the sonar number, in an array called *sonar_table[ ]*. This array can be accessed from any file that includes the **<sonar.h>** header file.

When a sonar interrupt occurs, the sonar interrupt handling routine, *SonarSysControl()* in **<sonar.c>** (Appendix A), is invoked. This function updates the *sonar_table[ ]* array for the sonars in the currently active sonar group. Of particular interest here is that it sets the update field to 1 (updated) and calls the *LinearFittingDecay()* function if the decay linear fitting flag is 1 (enabled). The update field is used to alleviate problems associated with the timing differences between the motion system and the sonar system interrupts. The *LinearFittingDecay()* function is used to update the linear fitted line representing the sonar data (i.e. the wall). This function is located in **<sonarmath.c>**. It does not actually do the linear calculations but is used to perform error checking before calling the *AddToSegmentDecay()* function, which does the work.

AddToSegmentDecay() is also located in **<sonarmath.c>** and, when called, adds the new sonar return to the linear data and recomputes the linear fitted line. The arguments to this function are the sonar number and the global position of the new sonar return. The sonar number is used as an index into the array *seg_data_decay[ ]* which is a static file scope array holding the *SEGMENT_RES_DECAY* structures for each sonar. The function's second argument is the point that will be added to the linear data. If the addition of the new point will create a data set with two or more points, *AddToSegmentDecay()* adds the new point to the existing linear moments using the procedure described by Equations (3.42 – 3.47), recalculates the centroid and secondary moments using Equations (3.26 – 3.29), recomputes the line representation parameters using Equations (3.34) and (3.35), and sets the usable field to 1 (true). If the new point is the first point in the data set, the function only calculates the linear moments and the usable field will remain 0 (false).

Additional functions are provided in **<sonarmath.c>** to access or set various components of the arrays of structures associated with the linear fitting with decay algorithm. Each of these functions takes the sonar number as one argument since they reference arrays that are indexed by the sonar number. Table 4 contains a list of these functions, the array/structure and field they access, and a short description.

2. Using Linear Fitted Data to Represent Wall

The sonar system interrupt handler ensures linear data is kept up to date for those sonars that have been enabled for linear fitting. (NOTE: From this point forward, any reference to linear fitting, linear data, etc. will mean data associated with the decay

version of the linear fitting algorithm.) Any function may obtain a current copy of the data by calling *GetSegmentDecay(sonarNum)*. This function returns a copy of the current *SEGMENT_RES_DECAY* structure associated with the specified sonar. Before using the data in this structure, it should be checked for usability by testing the usable flag. If usable, the r and α values specify the line representing the data points. These values specify the distance and normal direction to the line from the global origin. The line representation specifies an undirected line. In order to represent a wall, the line must be converted to a directed line, such as a line configuration.

FUNCTION	Array Structure	Field	Purpose
EnableLinearFittingDecay( )	sonar_table[ ] SONARD	decayFitting	Sets decayFitting to 1 (enabled)
DisableLinearFittingDecay( )	sonar_table[ ] SONARD	decayFitting	Sets decayFitting to 0 (disabled)
GetSegmentDecay( )	seg_data_decay[ ] SEGMENT_RES_DECAY	Entire structure	Returns a copy of the structure
SetDecayFactor( )	seg_data_decay[ ] SEGMENT_RES_DECAY	decayFactor	Sets decayFactor to a specified value
ResetMomentsDecay( )	seg_data_decay[ ] SEGMENT_RES_DECAY	Entire structure	Resets all fields to default values and usable to 0 (false)

Table 4. Access and Set Functions of Linear Fitting w/ Decay Algorithm.

To convert the undirected line to a line configuration representing the wall, values for the x and y coordinates, orientation (β), and curvature (κ) must be found. Since it is a

line, its curvature will be zero. The x and y coordinates are easily converted from r and α through the relations:

$$x = r\cos\alpha \tag{4.1}$$

$$y = r\sin\alpha \tag{4.2}$$

The wall's orientation (β) depends on the robot's orientation (θ). The two possible values for β are $(\alpha + \pi/2)$ and $(\alpha - \pi/2)$. Figure 16 shows examples of both situations.

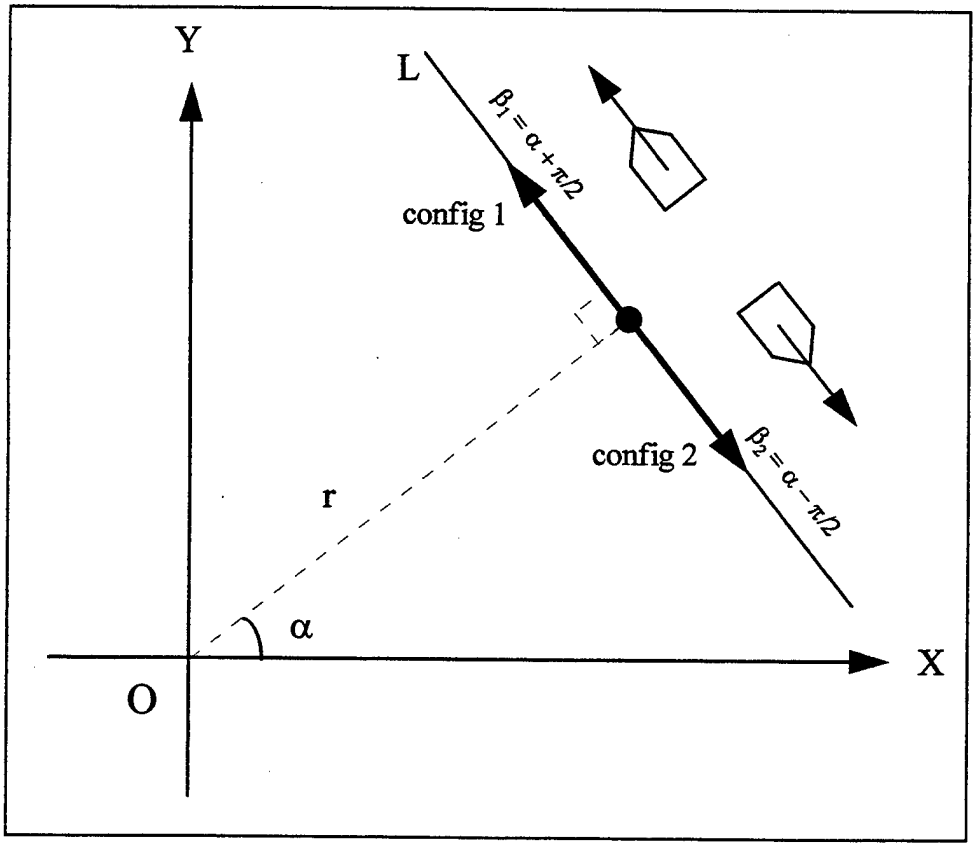


Figure 16. Two Possible Values of Wall Angle (β).

To determine the proper choice, one of these values is compared to θ. During wall following, the robot's orientation will be close to the wall's orientation and definitely

less than 90°. Therefore, the correct choice is found by comparing the magnitude of the difference between the two possible values and the robot's orientation:

$$\beta = \alpha - \frac{\pi}{2}, \ if \left| \Phi\left(\left(\alpha - \frac{\pi}{2}\right) - \theta\right)\right| < \frac{\pi}{2}$$
$$\beta = \alpha + \frac{\pi}{2}, \ otherwise \tag{4.3}$$

where $\Phi(\ )$ is the normalization function for angles.

The wall configuration has now been extracted from the sonar data and can be defined as (x, y, β, 0).

3. Following the Wall

In order to follow a wall using sonar, the procedure described in the last section is implemented each time new sonar data is available. In the wall following motion mode, the *wallHugRule()* function is called during every motion control cycle. This function is responsible for maintaining the current path element and calculating the commanded velocities that will keep the robot on this path.

Since the motion interrupt handler runs approximately five times between sonar interrupts when one sonar group is active, calculating a new configuration for the wall will only occur when new sonar data is available. Each time *wallHugRule()* is called, it first checks if the sonar data has been updated. If not, the current wall configuration remains unchanged. If new sonar data has been received, then the function calculates the new wall configuration using the procedure described in the last section.

The current path is calculated using the wall configuration and the specified offset distance. As stated in Section B.2, the offset value is passed into the motion system by the initialization function *trackWall( )*, through the radius field of the path element. The first time *wallHugRule( )* is invoked, the offset value is stored in the function scope static variable *offset*. The current path is computed by:

$$current\ path = wall\ config \circ (0, offset, 0, 0) \tag{4.4}$$

This places the current path parallel to the wall and on the appropriate side at the correct distance since *offset* is negative for left wall following and positive for right wall following.

The commanded velocities are found by using the function *steerByWall( )*. This function is called with the robot's configuration, the wall configuration, and the offset. It returns a steering function value which is used to calculate the commanded velocities. It is identical in functionality to the normal *steer( )* function for a line. It uses the distance to the wall minus the offset distance as the distance feedback and the difference between the robot's orientation and the wall's orientation as the angular feedback.

The process described in this section occurs during each motion control cycle and produces the desired effect of following a wall with a sonar. The "straightness" of the wall can vary slightly without loss of wall following ability because the linear fitting with decay algorithm makes timely adjustments to the linear representation of the wall and the robot adjusts its path accordingly.

D. ORTHOGONAL WALLS AND OBSTACLE AVOIDANCE

Once the straight wall following algorithm was created, it was used to expand the solution scope to include following walls with orthogonal turns. To solve this problem the robot uses the wall following algorithm while following straight wall segments until a turn is sensed. Turns are either concave (inside) or convex (outside). When a turn is required, the robot stops executing the wall following algorithm and enters a multi-phase process using lines, circular arcs, and the normal *steer()* function to navigate inside or around the corner of the wall. When the robot is abreast of a new wall, it returns to the wall following mode.

The sections that follow explain how turns are sensed, how the turn is accomplished, and how the determination is made to return to wall following.

1. Concave Corners

Concave corners are sensed by Yamabico's forward looking sonar (*S000)*. The criteria used to start an inside turn comes from the neutral switching point idea discussed in Chapter III (Section B.8). When the robot is following a wall and will make an inside turn to the next wall, the next path will 90° to the left or right of the current path. Figure 17 depicts this situation when the robot is in the left wall following mode.

The next path will also be offset from the next wall toward the robot's current position. In normal path tracking mode, the neutral switching point occurs when the steering function for the current line equals the steering function for the next line. Because the difference in angles between the current and next segment is known, an

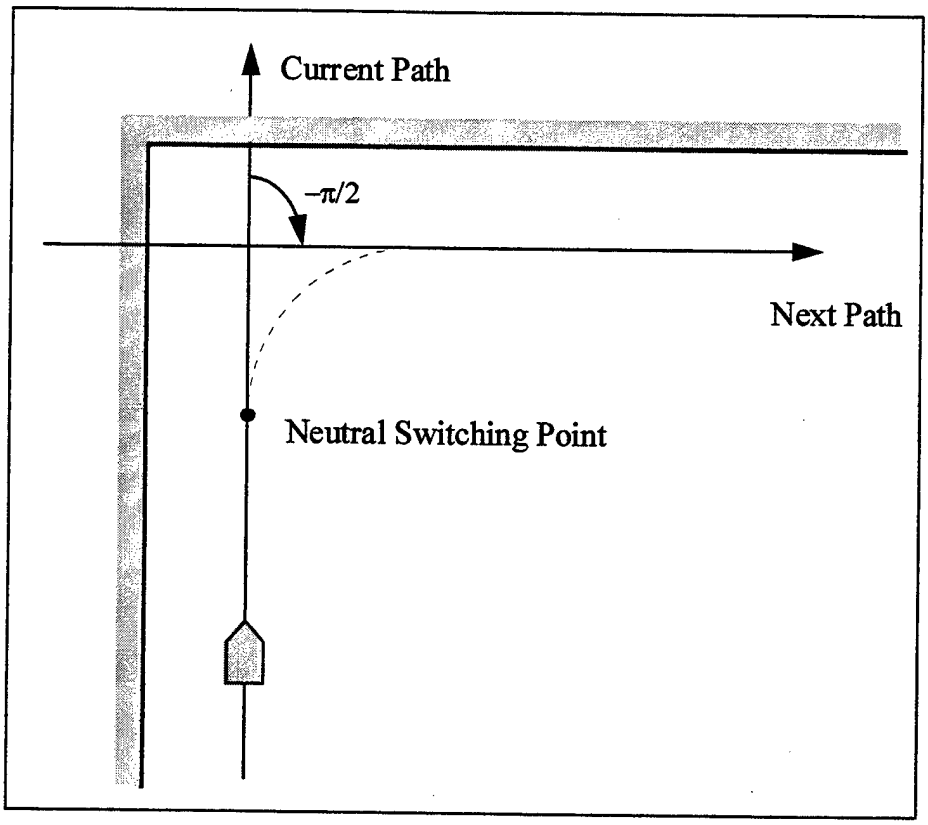


Figure 17. Neutral Switching Point from Current Path to Next Path.

expression can be found for the optimal distance to start a turn. This is accomplished by equating the two steering functions and solving for distance. The expression yielded by this was solved by Dr. Kanayama [2]:

$$d = 3\alpha\sigma \tag{4.5}$$

where d is the optimal turn distance (signed), α is the angular difference (π/2), and σ is the sigma value used in the turn.

The forward sonar is used to sense when this distance occurs. However, since the next path will be offset from the wall and the sonar senses the wall, not the path, the distance at which the robot should start its turn is:

$$d = 3\frac{\pi}{2}\sigma + |offset| \tag{4.6}$$

When the forward sonar senses a distance less than or equal to this value, the robot stops following the current wall (turns the side sonar off), computes an approximate next path line (L), sets the current path to L, and tracks the new current path using the normal *steer( )* function. This next line is computed relative to the robot's configuration by:

$$L = robot\ config \circ \left(dist000 - |offset|, 0, \Delta\theta, 0\right) \tag{4.7}$$

where dist000 is the forward sonar distance to the wall and $\Delta\theta$ is $\pm\pi/2$ depending on which side the wall is on.

While in the turn toward the new current path, the robot orientation is compared to that of the current path. Once this difference is less than a specified value (~10°), the robot is in a transition area. The side sonar is turned on and linear data for that sonar is reset via *ResetMomentsDecay( )*. While in this transition area, the robot continues to track the current path until sufficient linear data has been built for the next wall.

When there are enough data points in the linear fitting algorithm, the robot returns to the wall following mode.

Figure 18 shows the phases of a concave turn. To implement these phases, additional code was incorporated in the function *wallHugRule( )*. Static flag variables were added to perform checks on turn phase status (in turn, in transition) and turn type (inside, outside). A non-static flag was also added to determine the track type (following wall, current path). This flag is set each time the function is called and is used to

distinguish which type of calculations to use, the wall following algorithm or the normal path tracking algorithm.

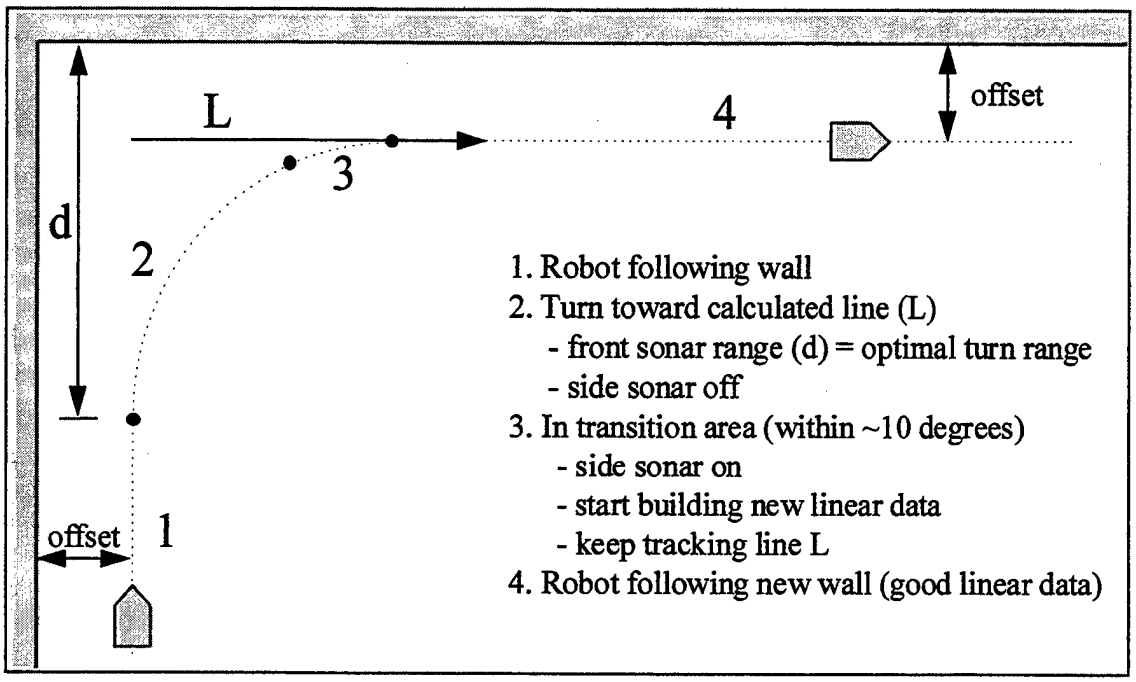


Figure 18. Phases of a Concave (Inside) Turn.

2. Convex Corners

Convex corners are sensed by the side sonar. As the robot proceeds along a straight wall, sonar distance readings vary only slightly. When the robot moves past a convex corner, the sonar distance measurement will jump from a fairly stable reading to a much larger distance. A convex turn is started when the difference between the current and previous sonar readings exceeds a certain threshold value.

The sequence of turn phases that occur during a convex turn are similar to those of the concave turn, but with an extra phase. When the threshold value is exceeded, the side sonar is turned off, the current path is changed to a circular arc, and the robot tracks

the current path around the corner. The circular arc is defined not only with a configuration, but also with a center position and radius. The arc's configuration is defined using the robot's current position and orientation with the curvature equal to the negative reciprocal of the offset value:

$$current\ path\ config = \left(robot\ x, robot\ y, robot\ \theta, -\frac{1}{offset} \right) \tag{4.8}$$

This gives the circular arc the proper direction (clockwise, counter-clockwise) with a radius equivalent to the offset distance. The current path structure also maintains a radius field and a center position field. These fields were unimportant in the wall following and concave turn implementations because only line paths were used, but must be set in this situation. The radius value is simply the reciprocal of the curvature. The center position is found by extracting the position from the following composition:

$$current\ path\ config \circ (0, -offset, 0, 0) \tag{4.9}$$

The robot tracks the current path (arc) for approximately 90° which should theoretically put it abreast the wall corner heading in the proper direction for the next wall. To allow for motion errors, however, the current path is set to a line configuration starting at the robot's current position with the robot's orientation. This causes the robot to track straight ahead from its current position.

The robot tracks the current path (line) for a specified distance to avoid the two situations depicted in Figure 19.

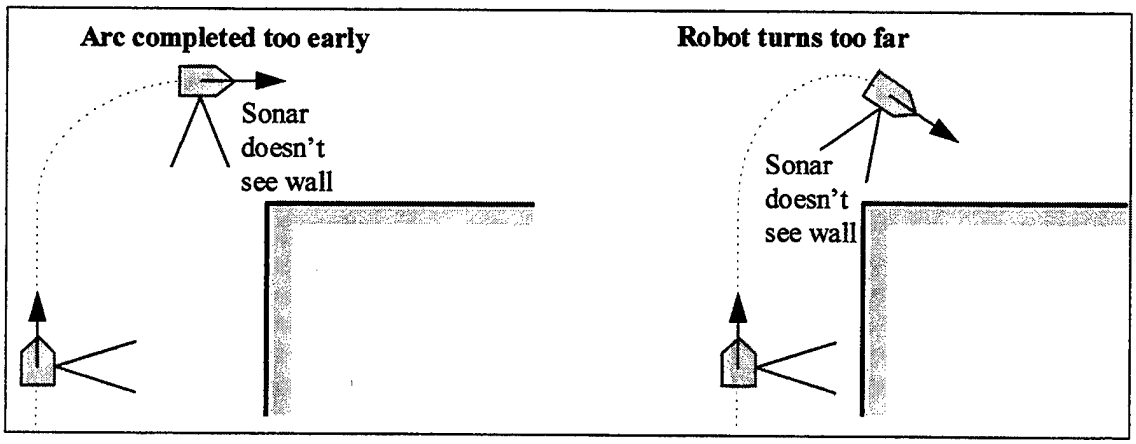


Figure 19. Undesirable Possibilities of a Convex (Outside) Turn.

To determine when the specified distance has been traversed, the relative configuration of the robot to the current path is computed by:

$$relative\ config = (current\ path\ config)^{-1} \circ robot\ config \tag{4.10}$$

When the x coordinate of this relative configuration is greater than the specified distance, the robot enters a transition area similar to the one specified in the concave turn. The side sonar is turned on and the linear data for that sonar is reset. While new linear data is being built, the robot continues to track the current path.

After sufficient linear data has been accumulated, the robot returns to the wall following mode and tracks the new wall.

Figure 20 shows the phases of a convex turn. To implement these phases, code was again added to the function *wallHugRule()*. The static flag variables continued to be used to perform turn phase and turn type checks. An extra value was included for the turn type flag to represent when the robot is tracking the circular arc. The track type flag

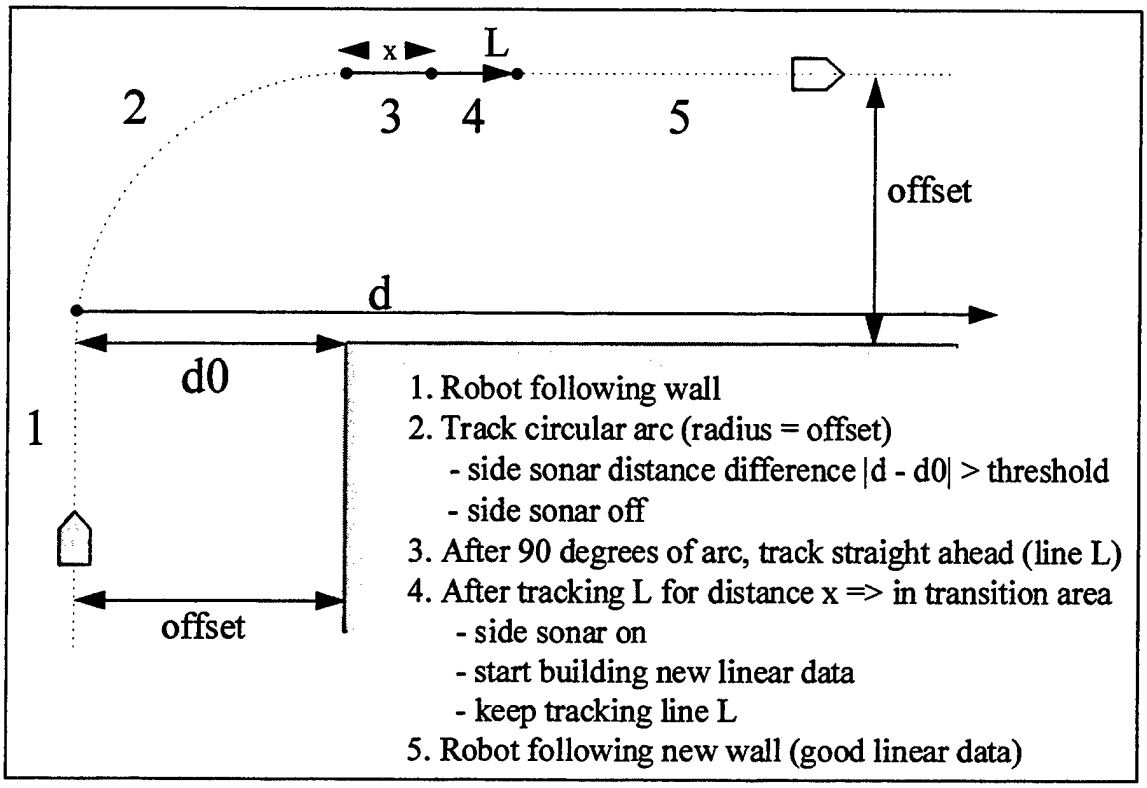


Figure 20. Phases of a Convex (Outside) Turn.

was not changed. Table 5 shows the values of the flags for the different phases in concave and convex turns.

3. Obstacle Avoidance

In the wall following mode, any obstacle in the robot's path is inherently close to the wall. By implementing the capability to navigate by following a wall and around orthogonal corners, obstacle avoidance is automatically included. The obstacle can be incorporated as part of the wall (Figure 21) and the robot uses the new capabilities to steer around it.

Phase	*inTurn*	*inTransition*	*turnType*	*trackType*
Concave:				
- wall following	0	0	--	0
- turning toward next wall	1	0	0	1
- in transition area	1	1	0	1
Convex:				
- wall following	0	0	--	0
- turning on arc	1	0	2	1
- tracking straight ahead	1	0	1	1
- in transition area	1	1	1	1

Table 5. Values of Flag Variables in *wallHugRule( )* During Phases of Turns.

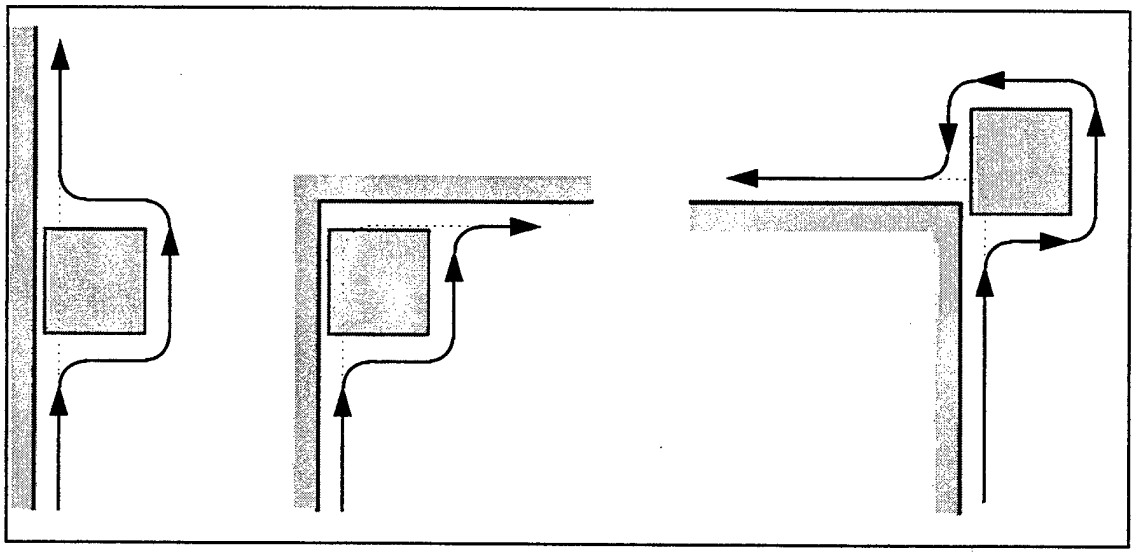


Figure 21. Examples of Obstacles in Wall Following Path.

4. Analysis of the *wallHugRule( )* Function

As previously stated, the *wallHugRule( )* function, located in **<motion.c>**, is the heart of the wall following motion mode. It is called by the *motionRules( )* function on

every motion cycle when the wall following mode is in use and returns the commanded velocities that keep the robot on the correct path. The function handles all the phases of wall following (straight wall following, concave turns, convex turns) discussed in this chapter. Because this function is so important, a detailed analysis of the code is warranted. The code for this function is listed in Appendix C with reference line numbers. In the analysis that follows, line numbers will be indicated by specific reference (e.g. line ##) or by a number in parentheses.

Line 9 is the beginning of the function header. It specifies that the function will return the commanded velocities (linear and rotational) in a structure called *VELOCITY*. The function accepts as arguments (line 10) the actual and commanded velocities used by the robot during the previous motion control cycle.

Lines 13-48 declare the variables used in the function. The *CONFIGURATION* variables are used to define and manipulate wall representation, offset , relative positioning , inverse, and temporary configurations during code execution. Linear fitted data from the sonar is extracted into the *curSeg* variable. Wall configuration components are stored in the static *wall___* variables defined in line 16 and the previous angle of the wall is kept in the static variable *prevWallTheta* (line 17). Line 19 defines a static variable to store the initial orientation value of a circular arc. The *deltaDistance* variable is used in the final computation of the commanded velocities and *deltaAngle* is used to determine which way the robot will turn in various situations depending on which side the wall is on. While in a turn, the robot's sigma value differs from the value used for straight wall following. *savedSigma* saves the old value while the robot turns.

pingedOnce and *pingedTwice* are flags used to ensure good history data from the sonar. *firstTimeCalled* is only true the first time the function is called. The next two lines (28, 29) declare variables that hold the side sonar number and sonar distances. *offset* stores the required offset distance the robot will maintain from the wall. This value is negative for left wall following and positive for right wall following. Lines 37-43 define the flag variables which are used to specify which motion state the vehicle is in. The first of these, *halted*, is used in collision detection. The next four are the turn phase indicators depicted in Table 5. The final three function variables are used in steering calculations. *jL* holds the steering function value, *kk* and *kk2* hold the current path's kappa and kappa squared values, and *k* is the steering feedback gain parameter.

After the variable declarations, the function first tests which type of wall following (left or right) is in use through the *currentPath* structure. This structure is a static file scope variable which stores the current path information during each motion control cycle. The switch statement in lines 53-66 assigns the appropriate values to *sonarNum* and *deltaAngle* depending on the wall following mode.

It was stated earlier that the offset value is passed into the motion control system through the radius field of *currentPath*. This takes place in lines 69-71. When the function is first called, the offset value is copied from *currentPath.Radius* into *offset*.

Line 74 extracts the linear fitted data from the sonar system and places it in *curSeg* using the **<sonarmath.c>** function *GetSegmentDecay( )*. Once this data available, lines 77-82 calculate the orientation of the wall using Equation (4.3).

Before any motion calculations take place, lines 86-93 check for a possible upcoming collision. If the forward sonar has been updated and the distance it measures is less than a predefined threshold, the *halted* flag is set, commanded velocities are set to zero, and the function returns.

To ensure good sonar data history, lines 96-104 determine if the side sonar has been updated (pinged) zero, one, or two times and sets the flags *pingedOnce* and *pingedTwice* appropriately.

The next section of code (112-247) performs the motion phase determination and adjusts the *currentPath* to keep the robot on course. The outer if statement breaks down the phases into blocks for wall following (112-180), in turn (183-230), and in transition area (233-247).

If the *inTurn* flag is not set, the robot is in the wall following mode, and the first code block is entered. The code in this block first sets the sonar distance variables. The front sonar distance is compensated for displacement from the robot center (115). The current and previous sonar readings are taken in raw form. These values are used in the nested if statement to check for turn indications. The if statement on lines 121, 122 first checks that the sonar has been updated and then uses the criteria described by Equation (4.6) to determine if the robot should begin an inside (concave) turn. When the criteria is met, the *inTurn* flag is set, the *turnType* is set to the value for an inside turn, and the current sigma value is saved and then set to a predefined value. Lines 127-131 update the *currentPath* configuration using the model described in Equation (4.7). Because the sigma value was changed, lines 132-135 update the steering parameters of *currentPath* to

reflect the change. Finally, because the robot is now in a turn, the *trackType* flag is set to 1 to signify that the normal *steer()* function should be used to track the current path. The next two lines finish the code block by resetting the sonar update field and disabling the side sonar.

If the criteria for an inside turn is not met, the code next checks for an outside turn. Lines 144-146 check if the sonar has been updated and has good history data, and if the difference between the current and previous returns exceeds a predefined threshold. If all these conditions are met, the robot will enter an outside turn. Lines 147-150 set the *inTurn* flag, save and reset the robot sigma, and set the *turnType* variable to the value 2 indicating the robot is on a circular path. At this point, *currentPath* is changed to a circular arc (155-163). The initial angle is saved for later comparison, the kappa value is calculated, and the steering feedback gains are set by lines 155-161. Note here that the value used for the radius is actually $\frac{3}{4}$*offset* instead of *offset*. This was a late change to the code but allows the robot to make a tighter turn around the corner. Lines 162, 163 set the configuration of *currentPath* as described by Equation (4.8). The center and radius of *currentPath* are set in lines 165-169 as prescribed by Equation (4.9) and its preceding paragraph. The next three lines (171-173) set the *trackType* to 1 (track current path), reset the sonar update field, and disable the side sonar.

If neither criteria (inside or outside turn) is met, the robot simply continues to follow the wall (lines 177, 178).

The brace on line 180 closes the first block of code. The second block of code begins on line 183 and handles the phase of motion when the robot is in a turn but not yet in the transition area.

If the robot is in an inside turn (*turnType* == 0) and the robot's orientation is within a predefined threshold of the current path (lines 188-190), then the *inTransition* flag is set, the side sonar is turned on, and the linear fitting data and *pinged___* variables are reset by lines 191-194.

If not on an inside turn, line 197 checks if the robot is on an outside turn (phase 3 in Figure 20). If so, then the procedure described by Equation (4.10) and its following paragraph is performed by lines 198-201. If the check in line 201 is passed, then the robot is in transition and lines 202-205 set the *inTransition* flag, turn on the sonar, and reset the linear data and *pinged___* variables.

When the robot is on the circular arc of an outside turn (*turnType* == 2), the first condition of the if statement on lines 211-215 is met. The next two conditions check if the robot's orientation has changed by (*deltaAngle* ± 3°) from either the circular arc's starting angle or the previous wall's angle. The angular error factor was empirically selected through testing. If either of these criteria is met, the robot proceeds straight ahead on the straight portion of an outside turn (*turnType* == 1). Lines 217-224 update *currentPath* and *turnType* accordingly.

For all three situations in which the robot is in a turn, the *trackType* flag is set to 1 (line 229).

The brace on line 230 closes the second block of code. The third and final block handles the motion phase when the robot is in the transition area of a turn and begins on line 233.

While in the transition phase, the linear fitting data is being built. When the data becomes usable, the sonar system will set the usable flag. Line 236 checks this flag, and if the data is usable, the robot will return to the wall following mode using the old sigma value (lines 237-240). If the data is not yet usable, the robot remains in the transition phase (line 245).

The brace on line 247 closes the third motion phase block of code. All that is left to do is to calculate the commanded velocities that will be returned.

The switch statement on lines 249-278 calculate the steering function value using either the *steerByWall( )* or *steer( )* function. The correct steering function is chosen by the status of the *trackType* variable. For the wall following mode (*trackType* == 0), the wall configuration is first defined. Lines 252, 253 calculate the wall's x and y coordinates using Equations (4.1) and (4.2). The wall's angle was already computed so line 255 uses these values to define *curWall*. Lines 259, 260 update *currentPath* using Equation (4.4) and line 263 retrieves the steering function value using *steerByWall( )*. Finally, the wall's current orientation is saved in *prevWallTheta* (line 266) to be available for future use.

If the *trackType* is 1, the current path will have already computed and the steering function value is found using the normal *steer( )* function (line 271), allowing the robot to track the current path.

After the default case for the switch statement (just a precaution) come the final lines in the function. Line 280 always sets *firstTimeCalled* to zero since the function is only called for the first time once. Line 282 uses a motion system function to calculate the linear portion of the commanded velocities and line 284 calculates the distance the robot has traveled over the last motion control cycle. This distance is combined with the steering function value, the robot's current curvature, and the commanded linear velocity to compute the required rotational velocity (lines 286, 287). Once the commanded velocities have been determined, the *wallHugRule( )* function is complete and the commanded velocities are returned to the calling routine.

E. USER PROGRAM

The previous sections described how the wall following mode is implemented and used to set robot on course. The user program, included in Appendix B, is used for operator interaction for setting up initial system parameters and for end of run condition checks. Before calling the *trackWall( )* function, the user program prompts the operator for system parameters such as whether sonar logging is on, the desired wall offset, the sigma value, the linear decay factor, and which side the wall will be on. The program uses this information to initialize corresponding parameters in the software system. The *trackWall( )* is then used to initialize the wall following mode. At this point the robot begins its motion, and the remaining portion of the user program monitors the robot's progress until end of run conditions are met. Once the robot has either halted or

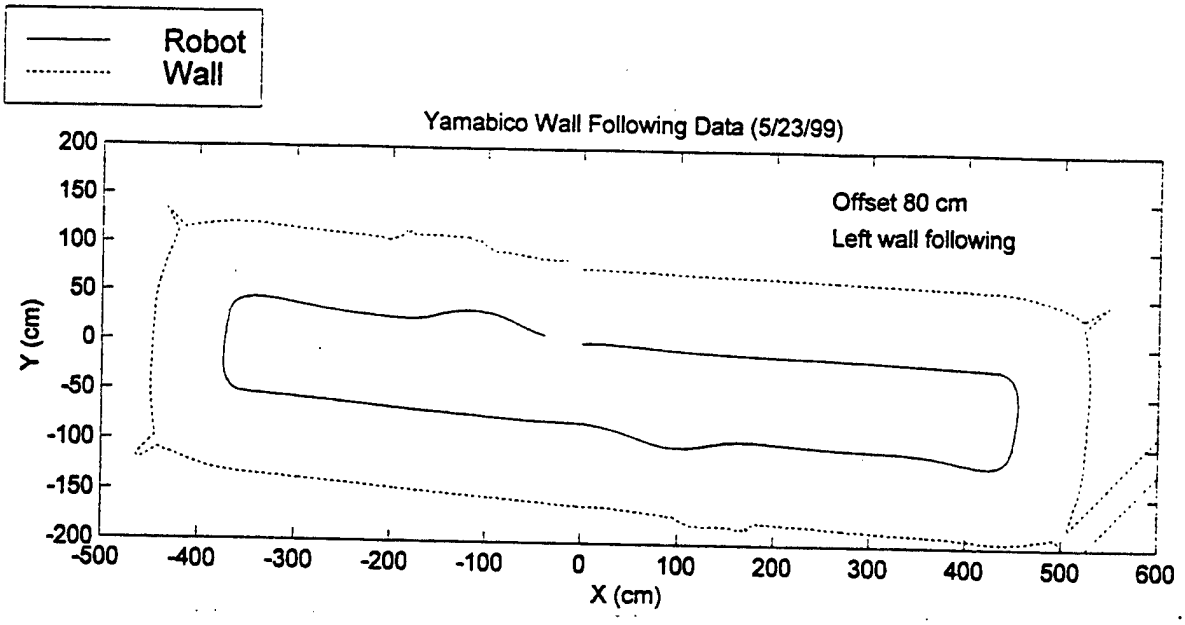
completed a complete circuit, the program shuts down the sonar system, stops the motion processing, and returns to the main program to download logged data.

V. THESIS RESULTS AND CONCLUSIONS

A. RESULTS

The goal of this thesis was to give an autonomous vehicle, such as Yamabico, the capability to navigate about a room or rooms composed of continuously connected orthogonal wall segments while avoiding obstacles in its path. The result of the research effort in implementing this behavior produced the desired effect. By using the new motion mode now incorporated in the MML software library, Yamabico can navigate around such a room in a safe, smooth, and efficient manner. Several tests were conducted to verify this fact and the data collected supports this conclusion.

Figure 22 shows actual data collected from Yamabico while traversing a hallway. The x and y axes are measured in cm. Both runs were completed using the left wall following mode. The first run used an offset of 80 cm while the run in the second run used a 50 cm offset. Both runs demonstrate the ability to maintain an offset distance while traversing a straight wall and smoothly negotiate both minor variations in the wall and concave corners. The angular tilt in the first figure is caused by initial positioning of the robot and errors associated with dead reckoning (DR) navigation. Even though these errors would be significant in normal path tracking mode, they are minor in the wall following mode since the robot is maintaining the appropriate distance from the reference wall, as indicated by the wall data being tilted by an equal amount.



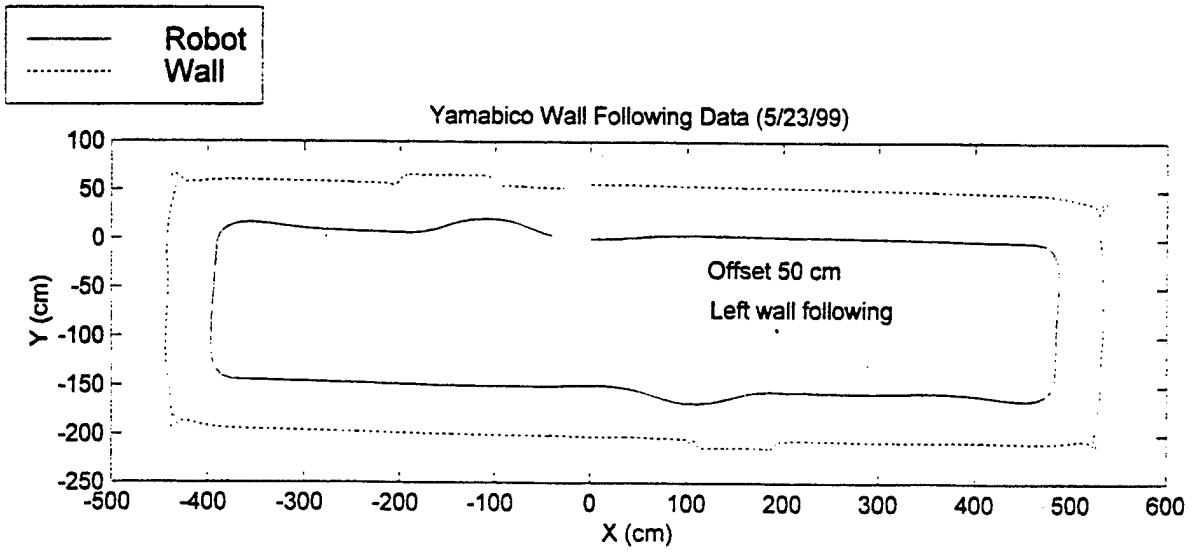


Figure 22. Left Wall Following with 80cm and 50cm Offsets.

To demonstrate the ability to negotiate a convex turn as well as obstacle avoidance, Figure 23 shows the motion and sonar data of a run starting in the corner of a room with convex corners, extending out and around an outside hallway containing an obstacle, and then back into the room completing the circuit. This run was performed using right wall following with an offset distance of 50 cm. The robot once again maintains the correct offset distance from straight wall segments while smoothly negotiating both concave and convex corners. As before, DR navigation errors accumulate during this extended run, indicated by the increasing tilt of the walls as the run progresses and the difference between the starting and finishing positions. The robot, however, maintains the correct offset distance from the reference walls throughout the run. The obstacle is treated as just another series of turns.

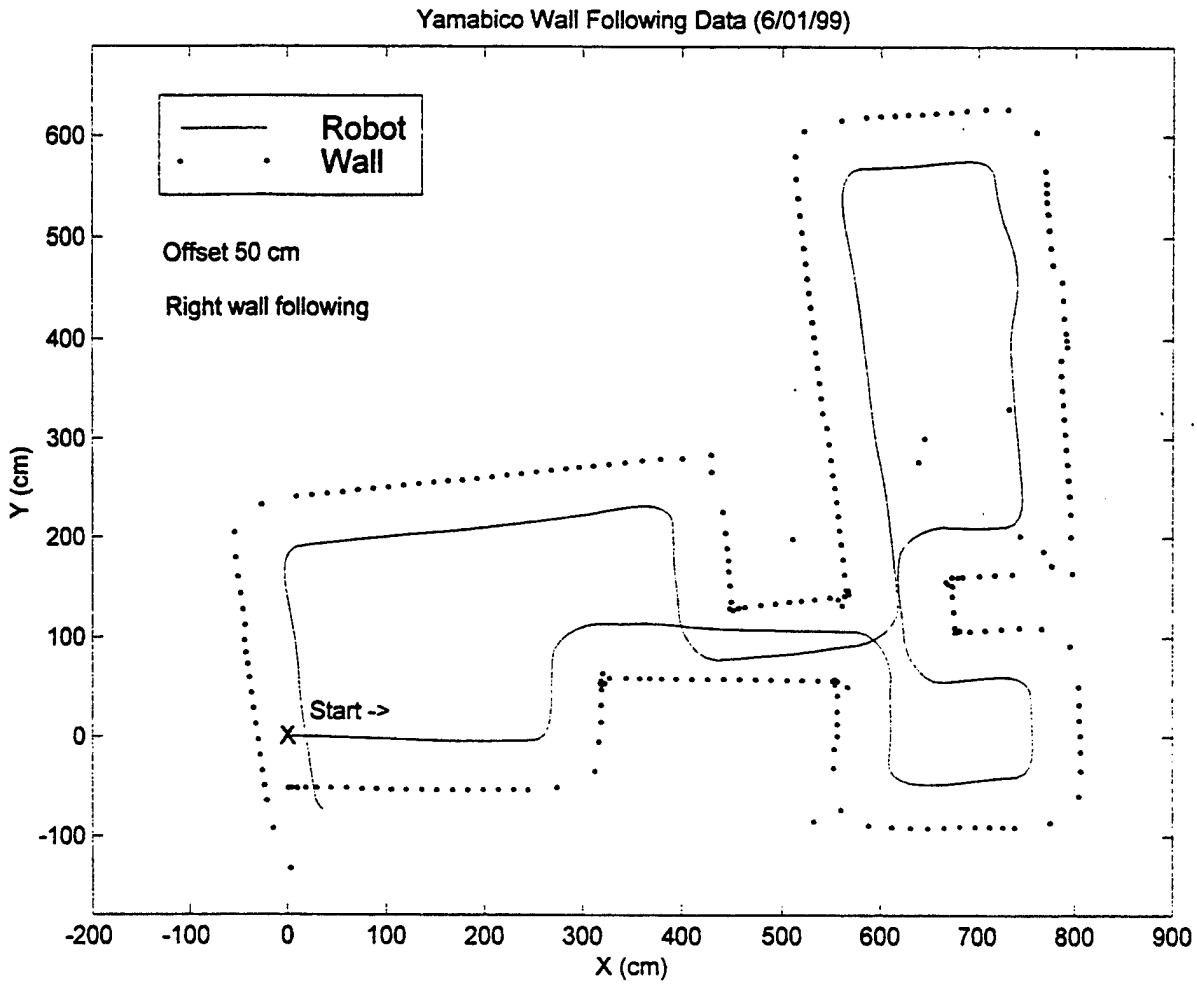


Figure 23. Complete Circuit of Two Rooms Using Right Wall Following with 50cm Offset.

With the successful implementation of a wall following motion mode, a solid foundation has been formed for the development of many advanced abilities.

B. FUTURE WORK

The work completed in this thesis was accomplished with fairly broad goals in mind and, although important, should be considered as a first step toward more complex behaviors. Some ideas for future work are presented in the following sections.

1. Starting Point

During the development of the wall following mode, the robot's starting point was always in close proximity to the first wall. For a more versatile behavior, the starting point should not have this limitation. An algorithm could be developed allowing the robot to start anywhere inside a room and search for the first wall. One possible implementation might be to have the robot proceed straight ahead until encountering a wall and then make a decision whether to turn right or left to enter the wall following mode.

2. Incorporate an S-turn Ability

While the robot is following a wall, minor variations in the wall cause the robot to adjust its course to maintain the offset distance. If the robot senses a large difference in side sonar distance between the current and previous readings, it enters a convex turn. There is a range of distances that is less than the threshold for convex turns but too large for the robot to compensate for. A possible fix to this problem is to incorporate an S-turn

when this situation arises. The robot could turn 45° toward the wall and then reverse this turn to again match the angle of the wall.

3. Other Than Orthogonal Turns

An obvious extension to the work done in this thesis would be to develop a more complex algorithm that would handle a wider variety of turn angles. During the development of the straight wall following algorithm, testing was performed on walls that varied in angle by values up to 20°. The robot compensated fairly well for angle differences of 15° degrees and below. Modifying the orthogonal turn algorithms to handle turns of 30° should result in a very robust behavior.

4. Better Collision Detection

The current collision detection mechanism is rather elementary. The robot uses its forward looking sonar to detect a situation where, by some error or miscalculation, it is too close to something directly ahead. This collision detection mechanism could be improved by use of additional sonars, such as those angled ±30° of the nose.

5. Very Small Obstacle Avoidance

The robot is capable of avoiding obstacles in its path as long as those obstacles are seen by the sensors. Because the forward looking sonar is positioned on the centerline of the robot, small obstacles that are positioned inside the sonar, but in the path of the wheel closest to the wall, will be hidden from the sensor and will cause a collision. The solution to this problem is not simple. One idea would be to use one of the quartering sonars as a warning detector and have the robot make adjustments to avoid a collision. Another idea would be to add forward looking sonars at the corners of the robot. This would very

difficult since it would require a hardware change and modification of the software-hardware interface.

APPENDIX A: MML FILES - MODIFIED OR IMPLEMENTATION RELATED

```
/*************************************************************************/
/*    File: constants.h                                                  */
/*                                                                       */
/*    last update                                                        */
/*       June 3, 1993 by Dave MacPherson                                 */
/*       Wed  04-27-94  Frank Kelbe - update for mml11                   */
/*       Sun  09-18-94  Cleaned up for clean mml11 version               */
/*       May, 1999  Dan Wells - added constants for linear fit w/ decay  */
/*                                                                       */
/*************************************************************************/
#ifndef __CONSTANTS_H
#define __CONSTANTS_H

   /* default motion logging filename */
#define     MOTIONLOGFILE     "motion.log"

/* there is no default sonar logging filename, because the type of logging
   must be specified for sonar logging, and the default filename is based
   on the type of logging to be performed */


   /* default motion logging buffer size (128K) */
#define     MOTIONLOGSIZE     0x20000

   /* Sonar logging buffer size (64K) */
#define     SONARLOGSIZE      0x10000

   /* the default IO buffer size (64K) */
#define     TRACELOGSIZE      0x10000



/*** system limits ***/

   /* max number of sequential instructions that can go in the buffer */
#define  INST_MAX             500

   /* max number of tracing/logging buffers in the system at once */
#define  MAXIOBUFFERS         20

#define  DEFAULT_PRECISION    6
#define  MAX_REAL_PRECISION   15
#define  MAX_INTEGER_DIGITS   19

   /* defines the length of time (in seconds) of the motion control cycle */
#define  MOTION_CONTROL_CYCLE 0.01


/* generic defined constants */

/* #define  NULL           0 */
/* old def of NULL above. Standard C def below */

#ifndef NULL
#define  NULL           ((void *)0)
#endif

#define  INFINITY       999999
#define  INFINITY0      99999
#define  PI             3.14159265358979323846
#define  DPI            6.28318530717958647692    /* PI * 2    */


/*******************************************************/
```

```
/*******************************************************/

/* Just info - not used anywhere - right now */

/*** machine limits ***/
#define  DBL_MIN        2.2250738585072014e-308
#define  DBL_MAX        1.7976931348623157e+308
#define  INFINITE       DBL_MAX
#define  INF            DBL_MAX

/* Numerical Constants */

#define HPI             1.57079632679489661923     /* PI/2      */
#define PI34            2.35619449019234492885     /* 3PI/4     */
#define PI4             0.78539816339744830962     /* PI/4      */
#define RAD             57.29577951308232087684    /* 180 / PI  */
#define DAR             0.017453292                /* PI / 180 */


/*********************************************************************/
/*  constants defination  for track.c                                */

#define S0      0.0
#define r2d     57.29577951
#define d2r     0.017453292
#define bufferMax 100
#define QPI                  PI/4.0

/*********************************************************************/
/*  constants for linear fitting with decay and wall following       */
/*  Added by Dan Wells, May 99                                       */

#define DEFAULT_DECAY        0.9f
#define DEFAULT_WALL_OFFSET  80     /* from robot center */
#define MIN_EDGE_OFFSET      10     /* from robot edge */
#define INSIDE_TURN_SIGMA    5.0f   /* for tighter inside turns */
#define OUTSIDE_TURN_SIGMA  10.0f   /* for outside turns */
#define COLLISION_THRESH     30     /* distance from front sonar */
#define TRANSITION_ANGLE     0.17453f  /* 10 degrees */
#define OUTSIDE_TURN_THRESH  80     /* diff betwn cur and prev sonar retrn */

#endif



/*****************************************************************************
   Author(s):   Scott Book
   Project:     Yamabico Robot Control System
   Date:        December 8, 1993
   Revised:     May 3, 1994 by Kevin Huggins
                May, 1999 by Dan Wells - wall following motion mode
   File Name:   definitions.h
   Environment: GCC ANSI C compiler for the motorola 68020 processor
   Description: This file contains standard definitions and data type
                declarations used throughout the reset of the MML system.
*****************************************************************************/

#ifndef __DEFINITIONS_H
#define __DEFINITIONS_H

/* Always include this because it is needed by most modules */
#include "constants.h"

typedef unsigned char  BYTE;
typedef unsigned short WORD;
typedef unsigned long  LONG;

typedef enum   {     NOMODE,
                     ENDMODE,
                     STOPMODE,
```

```
                    TRACKMODE,
                    TRACKSMODE,
                    TRACKRWALLMODE,
                    TRACKLWALLMODE,
                    ROTATEMODE,
                    PARAMODE,
                    BIDIRMODE,
                    KSPIRALMODE,
                    FOLLOWMODE,
                    REGIONMODE,
                    PCMODE
                } MODE;

typedef enum    {   NOCLASS,
                    LINECLASS,
                    CIRCLECLASS,
                    BLINECLASS,
                    NOSTOPBLINECLASS,
                    BIDIRCLASS,
                    LEFT,
                    RIGHT,
                    CENTER,
                    CCWLEFT,
                    CCWRIGHT,
                    CWLEFT,
                    CWRIGHT

                } CLASS;

typedef enum    {   INITMOTION,
                    INTERMOTION,
                    INTRAMOTION,
                    FINALMOTION
                } STAGE;


typedef enum {FALSE = 0, TRUE} BOOLEAN;


typedef struct {
  MODE      mode;
  CLASS     class;
} PATH_TYPE;


typedef struct {
   double   Linear;
   double   Rotational;
} VELOCITY;


typedef struct {
   double   X;
   double   Y;
} POINT;


typedef struct{
   POINT    Posit;
   double   Theta;
   double   Kappa;
} CONFIGURATION;


typedef struct{
  POINT           Focus;
  CONFIGURATION   Directrix;
} PARA;

typedef struct{
  CONFIGURATION   config;
```

```
  PATH_TYPE       pathType;
} PATH_ELEMENT;


typedef struct{
   POINT    LeftStart;
   POINT    LeftEnd;
   double   LeftOrient;
   POINT    RightStart;
   POINT    RightEnd;
   double   RightOrient;
} REGIONEDGE;

typedef struct  Image{
      POINT           coord;
      int             ImageType;
      double          Orient;
      double          ClosedDist;
      }         Image;

/* Error Codes */

typedef enum   {
      NOERROR,
      ECODE0,  /* configurations too close         */
      ECODE1,  /* erroneous goal configuration orientation  */
      ECODE2,  /* SSTOP function detected in moving state   */
      ECODE3   /* undefined instruction class         */
   }  ERROR_CODE;

/*ADDED*/
typedef struct {
  MODE            Type;
  CONFIGURATION   config;
  double          a;
  double          b;
  double          c;
  double          Radius;
  POINT           Center;
  double          speed;
  int             direction;
} LINE;

#endif



/*****************************************************************************
   Author(s):   Kevin Huggins
   Project:     Yamabico Robot Control System
   Date:        April 14, 1994
   Revised:     May 3, 1994
   File Name:   geometry.c
   Environment: GCC ANSI C compiler for the motorola 68020 processor
   Description: This file contains the standard geometry functions that are
                called by several functions.
*****************************************************************************/

#include "definitions.h"
#include "math.h"
#include "geometry.h"
#include "motion.h"
#include "stdiosys.h"
#include "constants.h"

/*****************************************************************************
  Function: eu_dis()
  Purpose:  Computes the Eucledian distance between two given points
  Parameters: double x1,y1,x2,y2
  Returns: double
```

```
  Comments:
*****************************************************************************/
double
eu_dis(double x1, double y1,double x2, double y2)
{
  return sqrt(((x1 - x2) * (x1 - x2)) + ((y1 - y2) * (y1 - y2)));
}


/*****************************************************************************
  FUNCTION:   norm()
  PARAMETERS: double angle     ---- the angle to normalize
  PURPOSE:    normalize the input angle between -PI and PI
  RETURNS:    double:     the normalized angle in radians
  COMMENTS:   This is the most common normalizing function used in the system
              This performs that same as norm() and normalize)() in MML10.
*****************************************************************************/
double
norm(double angle)
     /*{
  while ((angle > PI) || (angle <= -PI))
    {
      if (angle > PI)
    angle -= DPI;
      else
    angle += DPI;
    }
  return angle;
}*/

{
  return(angle - DPI*(ceil((angle + PI)/DPI) - 1.0));
}



/*****************************************************************************
  FUNCTION:   positiveNorm()
  PARAMETERS: double angle     ---- the angle to normalize
  PURPOSE:    normalize the input angle between 0 and 2PI
  RETURNS:    double:     the normalized angle in radians
  COMMENTS:   Same functionality as pnorm() in MML10
*****************************************************************************/
double
positiveNorm(double angle)
{
  while ((angle >= DPI) || (angle < 0))
    {
      if (angle >= DPI)
    angle -= DPI;
      else
    angle += DPI;
    }
  return angle;
}

/*****************************************************************************
  FUNCTION:   negativeNorm()
  PARAMETERS: double angle     ---- the angle to normalize
  PURPOSE:    normalize the input angle between -2PI and 0
  RETURNS:    double:     the normalized angle in radians
  COMMENTS:   Same functionality as nnorm() in MML10
*****************************************************************************/
double
negativeNorm(double angle)
{
  while ((angle > 0) || (angle <= -DPI))
    {
      if (angle > 0)
    angle -= DPI;
      else
```

```
    angle += DPI;
    }
  return angle;
}

/*****************************************************************************
  FUNCTION:   normPIover2()
  PARAMETERS: double angle     ---- the angle to normalize
  PURPOSE:    normalize the input angle between -PI/2  and PI/2
  RETURNS:    double:   the normalized angle in radians
  COMMENTS:   This was designed specifically for parabola calculations
              By Mahmoud Wahdan 3/12/96
*****************************************************************************/
double
normPIover2(double angle)
{
  while ((angle > PI/2) || (angle <= -PI/2))
    {
      if (angle > PI/2)
        angle -= PI;
      else
        angle += PI;
    }
  return angle;
}

/*****************************************************************************
  FUNCTION:   signedDist()
    PARAMETERS: CONFIGURATIONS directerix
                POINT  focus
  PURPOSE:    to calculate the size(signed) distance between a point(focus) and
              a configuration (directrix).
    RETURNS:    double:  the signed difference
    COMMENTS:   This was designed specifically for parabola calculations
              By Mahmoud Wahdan 3/12/96
*****************************************************************************/
double
signedDist(POINT pt, CONFIGURATION config)
{
    return (-(pt.X - config.Posit.X) * sin(config.Theta) +
                                  (pt.Y - config.Posit.Y)* cos(config.Theta));
}

/*****************************************************************************
  FUNCTION:   defineConfig()
  PARAMETERS: double x,y,theta,kappa        --The values that define a
                                              configuration
  PURPOSE:    To allocate nad assign a configuration
  RETURNS:    CONFIGURATION:      a configuration
  COMMENTS:   Was called def_configuration() in MML10
*****************************************************************************/
CONFIGURATION
defineConfig(double x,double y,double theta,double kappa)
{
  CONFIGURATION newConfig;

    newConfig.Posit.X = x;
    newConfig.Posit.Y = y;
    newConfig.Theta = theta;
    newConfig.Kappa = kappa;

  return newConfig;
}


/*****************************************************************************
  FUNCTION:   defineParabola()
  PARAMETERS: double xf,yf           ---defines the focus
              double xd, yd, thetad  ---defines the directrix
  PURPOSE:    To allocate assign a parabola
  RETURNS:    PARA: a parabola type
```

```
  COMMENTS:   Was called def_parabola() in MML10
*****************************************************************************/
PARA
defineParabola(double xf,double  yf, double xd,
                 double yd, double td)

{
    PARA newPara;

    newPara.Focus.X = xf;
    newPara.Focus.Y = yf;
    newPara.Directrix.Posit.X = xd;
    newPara.Directrix.Posit.Y = yd;
    newPara.Directrix.Theta = td;
    newPara.Directrix.Kappa = 0.0;
    return newPara;
}

/*****************************************************************************
  FUNCTION:   reverseOrientation()
  PARAMETERS: CONFIGURATION original  -- the original configuration
                                     orientation changed by 180 degrees
  PURPOSE:    To reverse the orientation of a given configuration
  RETURNS:    CONFIGURATION: the reversed  configuration
  COMMENTS:   Was called negate() in MML10
*****************************************************************************/
CONFIGURATION
reverseOrientation(CONFIGURATION original)
{
  CONFIGURATION reversed;

  reversed.Posit.X = original.Posit.X;
  reversed.Posit.Y = original.Posit.Y;
  reversed.Theta = norm(original.Theta + PI);
  reversed.Kappa = original.Kappa;
  return reversed;

}

/*****************************************************************************
  FUNCTION:   findSymConfig()
  PARAMETERS: double a  -- distance from either point to the intersection of
                           both lines determined by the two configurations
         double alpha --The angular difference between both orientations
  PURPOSE:    This function finds the symmetric configuration of a configuration
              described by alpha and a above.
  RETURNS:    CONFIGURATION: symConfig --the symmetic configuration
  COMMENTS:   Was called def_sym() in MML10
              One drawback to this function is that it is not possible to
              represent a symmetric configuration whose alpha is equal to PI.
                find_symConfig1() is used to cover these situations
*****************************************************************************/
CONFIGURATION
findSymConfig(double a, double alpha)
{
  return defineConfig(a * (1.+ cos(alpha)), a * sin(alpha), alpha, 0.0);
}

/*****************************************************************************
  FUNCTION:   findSymConfig1()
  PARAMETERS: double d  -- distance from the origin (base configuration) to
                           the symmetric configuration.
         double alpha --The angular difference between both orientations
  PURPOSE:    This finds the symmetric configuration of a configuration
              described by alpha and a above.
  RETURNS:    CONFIGURATION: sym_config --the symmetic configuration
  COMMENTS:   Was called def_sym1() in MML10
              This function overcomes the drawback of the original
              find_sym_config() of not being able to handle the situation when
              alpha equals PI
*****************************************************************************/
```

```
CONFIGURATION
findSymConfig1(double d, double alpha)
{
  double beta = alpha/2;

  return defineConfig(d * cos(beta), d * sin(beta), alpha, 0.0);
}

/*****************************************************************************

  FUNCTION:   inverse()
  PARAMETERS: CONFIGURATION original --the original configuration
                                       in global coordinates
  PURPOSE:    To calculate the inverse of a given configuration
  RETURNS:    CONFIGURATION: the inversed configuration
                             such that;
                             original * inversed = Identity
  COMMENTS:   None
*****************************************************************************/
CONFIGURATION
inverse(CONFIGURATION original)
{
    CONFIGURATION inversed;

  inversed.Posit.X = -original.Posit.X * cos(original.Theta) -
                     original.Posit.Y * sin(original.Theta);

  inversed.Posit.Y =   original.Posit.X * sin(original.Theta) -
                       original.Posit.Y * cos(original.Theta);

  inversed.Theta = -original.Theta;
  inversed.Kappa = -original.Kappa;

  return inversed;

}

/*****************************************************************************
  FUNCTION:   compose()
  PARAMETERS: CONFIGURATION *first  -- pointer to the first configuration
                            *second -- pointer to the  second configuration
  PURPOSE:    To calculate the composition of the first and second
              configurations
  RETURNS:    CONFIGURATION: configuration which is the
              composition of the first and second configurations
  COMMENTS:   A typical example of the usage of this function is to determine
              the goal position of a configuration in global coordinates.  In
              such an example, the first argument would be the original
              configuration and the second argument would be the goal
              configuration in the original configuration's local coordinate
              system.  The resultant third argument would then be the goal
              configuration in global coordinates.Was called comp() in MML10
  LAST UPDATE: 10/25/94 Chien-Liang Chuang
*****************************************************************************/
CONFIGURATION
compose(CONFIGURATION * first, CONFIGURATION * second)
{
  CONFIGURATION third;
  double x,y, theta;
  double xx,yy,tt;

  x = second->Posit.X;
  y = second->Posit.Y;
  theta = first->Theta;


  xx = cos(theta) * x - sin(theta) * y +  first->Posit.X;
  yy = sin(theta) * x + cos(theta) * y +  first->Posit.Y;

  tt = first->Theta + second->Theta;/* no need to normalize  10/25/94 Chuang */
```

```
  third.Posit.X = xx;
  third.Posit.Y = yy;
  third.Theta = tt;
  third.Kappa = first->Kappa;

  return third;
}


/*****************************************************************************

  FUNCTION:   circularArc()
  PARAMETERS: CONFIGURATION length  --the arc length
                            alpha   --the end orientation
                            config  --pointer to the resultant configuration
  PURPOSE:    Given the arc length and alpha, to calculate the final
              configuration
  RETURNS:    CONFIGURATION: pointer to the final configuration
  COMMENTS:   The main purpose of this function is to be used in conjunction
              with compose() to form a new next(). In this case, length would
              actually be delta-s and alpha would be delta-theta.
              Circular_arc() would determine the configuration after the incre-
              mental move in the local coordinate system of the original
              configuration. Then compose() would take the original
              configuration (in global coordinates) and the incremental
                configuration (in local coordinates) to determine the
              incremental configuration in global coordinates.
*****************************************************************************/
CONFIGURATION
circularArc(double length, double alpha)
{

  double alpha2,alpha4;

  alpha2 = SQR(alpha);
  alpha4 = SQR(alpha2);
  return defineConfig((1- alpha2/6.0 + alpha4/120.0) * length,
                (0.5 - alpha2/24 + alpha4/720.0) * alpha * length,
                alpha, 0.0);
}



/*****************************************************************************
  Function  : orthogonalize()
  Purpose   : To set the input orientation to the nearest angle of a (integer)
              multiple of PI/2 (orthogonal orientation)
  Parameters: double theta
  Returns   : orthogonal orientation
  Comments:
*****************************************************************************/
static double
orthogonalize(double theta)
{
 if (theta >= 0.0)
   return (((int) ((theta + QPI) / HPI)) * HPI);
 else
   return (((int) ((theta - QPI) / HPI)) * HPI);
}


/*****************************************************************************
  Function  : computeTheta()
  Purpose   : To compute the orientation from a point to another
  Parameters: POINT p1, POINT p2
  Returns   : the orientation (type double)
  Comments:
*****************************************************************************/
static double
computeTheta(POINT p1, POINT p2)
  {
```

```
   return atan2(p1.Y - p2.Y, p1.X - p2.X);
  }


/*************************************************
  Function  : areaTriang()
  Purpose   : To determine the area of any triangle.
  Parameters: Point p1          the first point
              Point p2          the second point
              Point p3          the third point
  Returns   : the area of triangle (type double)
  Comments  : July 15, 1996  Mahmoud Wahdan
*************************************************/
static double
areaTriang(POINT p1, POINT p2, POINT p3)
{
      double          area;

      /* calculate the area of the triangle */
      area = 0.5*((p2.X - p1.X) * (p3.Y - p1.Y) -
             (p3.X - p1.X) * (p2.Y - p1.Y));

        return area;
}

/*****************************************************************************
  Function  : depth()
  Purpose   : To compute the depth of a point along a line defined by the
              orientation.
  Parameters: POINT p, double alpha -- a point and it's orientation
  Returns   : the depth of the point
  Comments:
*****************************************************************************/
static double
depth(POINT p, double alpha)
{
  return  p.X * cos(alpha) + p.Y * sin(alpha);
}


/*****************************************************************************
  Function: Psi_function()
  Purpose:  Computes the Psi function of two given points
  Parameters: POINT p1,p2
  Returns: double
  Comments:
*****************************************************************************/
double
Psi(POINT p1,POINT p2)

{
  if ( p2.Y - p1.Y == 0.0 && p2.X - p1.X == 0.0)
    return 0.0;
  else
    return atan2(p2.Y - p1.Y, p2.X - p1.X);
}


/*****************************************************************************
  Function: distance()
  Purpose:  Computes the  distance between two given points
  Parameters: POINT p1,p2
  Returns: double
  Comments:
*****************************************************************************/
double
distance(POINT p1,POINT p2)

{
  double X, Y;
```

```
  X = p1.X - p2.X;
  Y = p1.Y - p2.Y;
  return sqrt( X*X  +  Y*Y );
}

/*****************************************************************************
  Function  : intersectLineToLine()
  Purpose   : To compute the intersection of two lines.
  Parameters: CONFIGURATION p1, CONFIGURATION p2 -- two lines
  Returns   : configuration of intersection
  Comments:
*****************************************************************************/
CONFIGURATION
intersectLineToLine(CONFIGURATION p1, CONFIGURATION p2)
{
 CONFIGURATION inter;

 inter.Posit.X = ( -cos(p2.Theta) *
                    (p1.Posit.X * sin(p1.Theta) - p1.Posit.Y * cos(p1.Theta))
                  + cos(p1.Theta) *
                    (p2.Posit.X * sin(p2.Theta) - p2.Posit.Y * cos(p2.Theta))
                 )
                 / sin(p2.Theta - p1.Theta);

 inter.Posit.Y = ( sin(p1.Theta) *
                   (p2.Posit.X * sin(p2.Theta) - p2.Posit.Y * cos(p2.Theta))
                  - sin(p2.Theta) *
                   (p1.Posit.X * sin(p1.Theta) - p1.Posit.Y * cos(p1.Theta))
                 )
                 / sin(p2.Theta - p1.Theta);

 inter.Theta = p1.Theta;  /* in some cases, maybe p2.t will be needed */
 inter.Kappa = 0.0;
 return(inter);
}
/*****************************************************************************
  Function  : order()
  Purpose   : To determine the order (i.e. CW or CCW) of three points
  Parameters: Point p1          the first point
              Point p2          the second point
              Point p3          the third point
  Returns   : int  CW if clockwise, CCW if counterclockwise
  Comments:
*****************************************************************************/
int
order(POINT p1, POINT p2, POINT p3)
{
      double          area;

      /* calculate the area of the triangle */
      area = ((p2.X - p1.X) * (p3.Y - p1.Y) -
             (p3.X - p1.X) * (p2.Y - p1.Y));

      if (area > 0.0)
        return (1);
      else if (area < 0.0)
        return (-1);
      else
        return (0);
}
```

```
/*****************************************************************************
   File Name:   motion.c
   Environment: GCC ANSI C compiler for the SPARC processor
   Description: This file provides the routines and data structures needed to
                provide the motion control capability for the robot. The file
                is divided into three sections The first is the private section
                containong the encapsulated data and functions. The second
                section is the control section. This section defines the
                routines required for motion control. The third section is the
                Immediate command section. This section defines MML's immediate
                commands. The routines in these last two sections can be
                accessed publicly.
   Last update: May 30, 1995  Midportion MP is added (modified July,20 1995)
                June 18,1995  symmetric path tracking is added (testing)
                July 21,1995  SingleregionMP() added, tested ok.
                July 29,1995  learing up the motion.c
                May, 1999     Wall following motion mode added, Dan Wells
*****************************************************************************/

/*#define TIMER*/

#include "definitions.h"
#include "pcdefs.h"
/*#include "PCdefs.h" */
#include "wheels.h"
#include <math.h>
#include "queue.h"
#include "motionlog.h"
#include "geometry.h"
#include "stdiosys.h"
#include "time.h"
#include "system.h"
#include "trace.h"
#include "motion.c.h"
#include "sonar.h"
#include "sonarmath.h"  /* Added: D Wells 4-12-99 */
#include "sonarcard.h"  /* Added: D Wells 5-16-99 */
#include "seqcmd.h"
#include "memsys.h"


/*****************************************************************************

   PRIVATE SECTION

   The following section defines the encapsulated definitions, data structures
   and prototypes used in the system.

*****************************************************************************/

/*************************************************************
   constant definition
*************************************************************/

#define HALFTREAD             26.20623239064972878310
#define DEFAULT_LIN_ACC       10.0
#define DEFAULT_ROT_ACC       0.25
#define DEFAULT_GOAL_VEL_LIN  30.0
#define DEFAULT_GOAL_VEL_ROT  5.0
#define SIGMA                 20.0


#define LEFT                  1
#define RIGHT                 0
#define ON                    1
#define OFF                   0
#define NIL             -1
#define CW                    -1
#define COL                   0
#define CCW                   1
```

```
/*************************************************************
   global variables (in motion.c file) declarations
*************************************************************/
BOOLEAN           Halted;

static double     deltaS;

static VELOCITY   Commanded;  /*commanded velocities */

static double     kappaCommanded; /* commanded kappa */

static CONFIGURATION vehicle;       /* local variable for
                                    vehicle configuration*/

static LINE  currentPath; /* holds the current path element values */

static VELOCITY   haltedVel;
static VELOCITY   goalVel,
                  goalAcc;

static double     desiredSigma,
                  totalDistance;

static int LoopTest;

static double     lambdaNew, lambdaOld;

static int        decFlag = FALSE;


/************* the following functions are for model based motion **********/

/*************************************************************
   The following static function declarations are the
   prototypes for the encapsulated functions.
*************************************************************/


      /* calculates the next commanded linear velocity value.  */
static double computeLinVelocity(double ActualVelocity,
                                 double LastCommandedVelocity);

      /* calles a motion rule function based on the mode of travel that the
         vehicle is in */
static VELOCITY motionRules(VELOCITY Actual, VELOCITY Commanded);


      /* calculates the distance remaining on a path to reach a configuration.
        Used with bline calculation. */
static double restOfPath(CONFIGURATION q1, LINE l1);

      /* determines whether the vehicle needs to decelerate.
         Used in bline calculation */
static int needsToDecelerate(double actualVelocity);


      /* determines whether it's time to process the next instruction */


void
setSteeringVar();

/****** Global variables for parabola tracking ******/

static  VELOCITY
stopRule(VELOCITY actual, VELOCITY commanded);

static  VELOCITY
trackRule(VELOCITY actual, VELOCITY commanded);
```

```
static  VELOCITY
wallHugRule(VELOCITY actual, VELOCITY commanded);

static  VELOCITY
rotateRule(VELOCITY actual, VELOCITY commanded);

double steer(CONFIGURATION robot, LINE line);
double
steerByWall(CONFIGURATION robot, CONFIGURATION curWall, double offset);


/*****************************************************************************

   MOTION CONTROL SECTION

   The following section defines the functions that provide access to the
   motion control system. These routines are public.

*****************************************************************************/

/*****************************************************************************
 Function InitMotion() initializes all of the private global variables
   in this module to the default values. It then calls SetTimer to
   program the 5th timer on serial board #1(the second serial board)to generate
   synchronous interrupts every 10ms. After the timer has been set up, the
   interrupt handling routine is made available to the system by the call to
   SetMotionInterruptHandler().
*****************************************************************************/

void
InitMotion(void)
{
   LoopTest = 0;
   Halted = FALSE;
   haltedVel.Linear = 0.0;
   haltedVel.Rotational = 0.0;
   decFlag = FALSE;

     /* Initialize motion related systems */
   InitSeqcmd();
   InitWheels();

     /* Initializes the distance. Updated every motion control cycle by deltaS*/
   setTotalDistanceImm(0.0);

     /* Initialize the goal velocities */
   setLinVelImm(DEFAULT_GOAL_VEL_LIN);
   setRotVelImm(DEFAULT_GOAL_VEL_ROT);

     /* Initialize the commanded velocities */
   Commanded.Linear = 0.0;
   Commanded.Rotational = 0.0;

     /* Initalize the linear and rotational acceleration */
   setLinAccImm(DEFAULT_LIN_ACC);
   setRotAccImm(DEFAULT_ROT_ACC);

     /* Initialize the size constant */
   setSigmaImm(SIGMA);


   /* add a part of trackline  */

     /* Initialize the commanded kappa  */
   kappaCommanded = 0.0;

    /* Initialize the clearance  */
   /*setClearanceImm(CLEARANCE);*/

     /* Initialize the vehicle configuration */
```

```
    vehicle.Posit.X = 0.0;
    vehicle.Posit.Y = 0.0;
    vehicle.Theta = 0.0;
    vehicle.Kappa = 0.0;

      /* Initalize the current path configuration */
    currentPath.Type = STOPMODE;
    lambdaOld = INF;

}

/*****************************************************************************
    Function MotionSysControl() is the interrupt handler workhorse.It is called
    from the assembly interrupt handler shell. Its first task is to update the
    change in position and orientation through calls to the module responsible
    for movement. It then uses this information in the motion control laws to
    derive the commanded linear and rotational velocities required for this
    motion control cycle. Finally, it passes these computed velocities back to
    the movement module for execution.
*****************************************************************/

void
MotionSysControl(void)
{
    double deltaTheta;

    VELOCITY Actual; /* variable used to hold the actual vehicle velocity */

#ifdef TIMER
    int   tick = getCount();
#endif

    UpdateMovement(); /* updates the distance traveled by both wheels--found in
                         Wheels.c */
    deltaS = GetDistanceTraveled(currentPath); /*returns the linear distance the
vehicle
                                                 has travelled between the last two calls
to
                                                 UpdateMovement()--found in Wheels.c */
                                                 /* updated by mahmoud wahdan Nov, 14 95
*/

    deltaTheta = GetOrientationChange(); /* returns the difference between the
                                            changes in the distance of the left
                                            and right wheels between the
                                            last two calls to UpdateMovement().
                                            Found in Wheels.c */
    totalDistance += deltaS;  /* Keeps track of the total distance traved
                                 by vehicle */

    /*update the vehicle's configuration based on the distance travelled
      during the last motion control cycle */;

    vehicle = localize(vehicle, deltaS, deltaTheta);

    /* next 2 lines calculate the actual velocity that robot maintained based
       on the distance travelled over the last motion control cycle. */

    Actual.Linear = deltaS / MOTION_CONTROL_CYCLE;
    Actual.Rotational = deltaTheta / MOTION_CONTROL_CYCLE;

    /* logs the values of the vehicle configuration. Data is
       written to a buffer during each motion control cycle and
       then downloaded to a file when the program ends.  LogMotion
       is found in motionlog.c */

    LogMotion(vehicle);


    /* motionRules returns the commanded velocities that will be
       used in the next motion control cycle.  Found in this module.*/
```

```
   Commanded = motionRules(Actual,Commanded);

   /* SetMovement() translates the commanded linear and
      rotational velocities into commanded velocities for each
      wheel.  Found in wheels.c */

   SetMovement(Commanded.Linear,Commanded.Rotational);

   /*transition();  reads next instruction if needed. Found in this module.*/

   /* Increments the "time" every motion control cycle for the
      various timer functions.  Found in time.c */

   clockTick();

   /* blinkLED is  used to control output from interrupt driven
      motion control system. It turns an LED on and off every
      second. Function found in this module.*/

   blinkLED();
}


/*****************************************************************************
  Function   : setRobotConfigImm()
  Purpose    : To set and update the robot configuration
  Parameters: CONFIGURATION NewConfig
  Returns    : void
  Comments   :
*****************************************************************************/
void
setRobotConfigImm(CONFIGURATION NewConfig)
{
   DisableInterrupts();

   vehicle.Posit.X = NewConfig.Posit.X;
   vehicle.Posit.Y = NewConfig.Posit.Y;
   vehicle.Theta = NewConfig.Theta;
   vehicle.Kappa = NewConfig.Kappa;

   EnableInterrupts();
}


/*****************************************************************************
  Function   : getRobotConfig()
  Purpose    : Retreives the current robot configuration
  Parameters: Pointer to a variable where the current values for the robot's
              configuration will be placed.
  Returns    : void
  Comments:
*****************************************************************************/
CONFIGURATION
getRobotConfig(void)
{
   CONFIGURATION temp;

   DisableInterrupts();
   temp = vehicle;
   EnableInterrupts();
   return temp;
}



/*****************************************************************************
  Function   : localize()
  Purpose    : Calculates the next configuration of the vehicle based on the
               distance that the robot travelled during the last motion
               control cycle
```

```
  Parameters: CONFIGURATION robot --from the last motion control cycle
              double deltaS -- linear distance travelled in last
                               motion control cycle
              double deltaTheta --angular change in the last motion
                                  control cycle
  Returns   : CONFIFURATION --of the vehicle based on the distance travlled
                              during the last motion control cycle
  Comments:
*****************************************************************************/
CONFIGURATION
localize(CONFIGURATION robot, double deltaS, double deltaTheta)
{
   CONFIGURATION tempRobot;

   tempRobot = circularArc(deltaS, deltaTheta);
   robot = compose(&robot, &tempRobot);
   robot.Kappa = kappaCommanded;

   return robot;
}


/*****************************************************************************
  Function  : computeLinVelocity()
  Purpose   : Calculates the linear component of the commanded velocity.
  Parameters: double ActualVelocity, double CommandedVelocity
  Returns   : new commanded linear velocity
  Comments:
*****************************************************************************/
static double
computeLinVelocity(double ActualVelocity, double CommandedVelocity)
{
    double          VelocityChange;
    double          goalSpeed;

    VelocityChange = goalAcc.Linear * MOTION_CONTROL_CYCLE;
    goalSpeed = goalVel.Linear / (1.0 + HALFTREAD * fabs(kappaCommanded));

    if (ActualVelocity < goalSpeed)
      CommandedVelocity = Min(CommandedVelocity + VelocityChange,
                              goalSpeed);
    else
      CommandedVelocity = Max(CommandedVelocity - VelocityChange,
                              goalSpeed);


    return CommandedVelocity;
}

/*****************************************************************************
  Function  : computeLinVelocityS()
  Purpose   : Calculates the linear component of the commanded velocity.
  Parameters: double ActualVelocity, double CommandedVelocity
  Returns   : new commanded linear velocity
  Comments:
*****************************************************************************/
static double
computeLinVelocityS(double ActualVelocity, double CommandedVelocity)
{
  double          VelocityChange;
  double          goalSpeed;

  VelocityChange = goalAcc.Linear * MOTION_CONTROL_CYCLE;

  if(decFlag || needsToDecelerate(ActualVelocity))
    {
      CommandedVelocity -= VelocityChange;
      if(CommandedVelocity <= 0.0)
    currentPath.Type = STOPMODE;
    }
```

```
    else
      {
        goalSpeed = goalVel.Linear / (1.0 + HALFTREAD * fabs(kappaCommanded));

        if (ActualVelocity < goalSpeed)
      CommandedVelocity = Min(CommandedVelocity + VelocityChange,
                              goalSpeed);
        else
      CommandedVelocity = Max(CommandedVelocity - VelocityChange,
                              goalSpeed);
      }
    return CommandedVelocity;
}

/*****************************************************************************
  Function  : restOfPath()
  Purpose   : calculates the remaining distance to the ending configuration for
  Parameters: void
  Returns   : the distance between current position to the STOPPING
  configuration
  Comments:
*****************************************************************************/
static double
restOfPath(CONFIGURATION q1, LINE l1)
{
  CONFIGURATION  q2 = l1.config;
  if (q2.Kappa == 0.0)
    return ((q2.Posit.X - q1.Posit.X) * cos(q2.Theta) +
        (q2.Posit.Y - q1.Posit.Y) * sin(q2.Theta));
  else
    if (q2.Kappa > 0.0)
      return (l1.Radius*
          positiveNorm(Psi(l1.Center,q2.Posit) -
                       Psi(l1.Center,q1.Posit)));
    else
      return (l1.Radius*
          negativeNorm(Psi(l1.Center,q2.Posit) -
                       Psi(l1.Center,q1.Posit)));

}


/*****************************************************************************
  Function  : needToDecelerate()
  Purpose   : To determine whether the robot needs to begin decelerating.  Such
              as in a bline function.
  Parameters: double actualVelocity (linear)
  Returns   : It returns 1 if it needs to decelerate.  Otherwise, it returns 0.
  Comments:
*****************************************************************************/
static int
needsToDecelerate(double actualVelocity)
{
   int   decelerate;

   decelerate = FALSE;

   if (2.0 * goalAcc.Linear * restOfPath(vehicle,currentPath)
       <= actualVelocity * actualVelocity)
     decelerate = TRUE;

   decFlag = decelerate;
   return decelerate;
}



/*****************************************************************************
  Function  : motionRules()
  Purpose   : To calculate the linear velocity and rotational velocity based
```

```
              on the type of motion that the robot is executing.
  Parameters: VELOCITY  actual, commanded
  Returns   : The  commanded linear and rotational velocities,
  Comments:
*****************************************************************************/
static          VELOCITY
motionRules(VELOCITY actual, VELOCITY commanded)
{
  switch (currentPath.Type) {
  case STOPMODE:
    commanded = stopRule(actual, commanded);
    break;

  case TRACKMODE:
  case TRACKSMODE:
    commanded = trackRule(actual, commanded);
    break;

  case TRACKRWALLMODE:
  case TRACKLWALLMODE:
    commanded = wallHugRule(actual, commanded);
    break;

  case ROTATEMODE:
    commanded = rotateRule(actual, commanded);
    break;

  default:
    break;
  }

  return commanded;
}

/*****************************************************************************
  Function  : stopRule()
  Purpose   : updates the commanded velocity to 0 (zero) to stop the robot
  Parameters: VELOCITY actual, commanded
  Returns   : The required linear velocity, rotational velocity
  Comments:
*****************************************************************************/
static  VELOCITY
stopRule(VELOCITY actual, VELOCITY commanded)
{
  /*if there is a next line in the buffer, take out as the current line*/

   commanded.Linear = 0.0;
   commanded.Rotational = 0.0;

  if(!IsBufferEmpty())
    {
      currentPath = GetLine();
    }

   return commanded;
}

/*****************************************************************************
  Function  : testNeutral()
  Purpose   : to determine if the robot is at the neutral switching point
  Parameters: None
  Returns   :
  Comments  :
*****************************************************************************/
int testNeutral(LINE nextLine, double lambdaNew )
{
  double angle;

  if (nextLine.config.Kappa == 0) {
    if (currentPath.config.Kappa == 0) {
      angle = norm(nextLine.config.Theta - currentPath.config.Theta);
```

```
      if (fabs(angle) < 0.001) {
        return TRUE;
      } else {
        return (angle*lambdaNew >= 0.0);
      }
    } else {
      return 0;
    }
  } else {
    return 0;
  }
}

/*****************************************************************************
  Function  : trackRule()
  Purpose   : To determine the linear and rotational velocities needed to put or
              keep Yamabico on the path.
  Parameters: VELOCITY actual, commanded
  Returns   : The required linear velocity, rotational velocity
  Comments:
*****************************************************************************/
static  VELOCITY
trackRule(VELOCITY actual, VELOCITY commanded)
{
  double deltaDistance, angle;
  LINE nextLine;

  if(!IsBufferEmpty())
    {
      nextLine = ReadNextLine();
      lambdaNew = steer(vehicle,nextLine);
      angle = fabs(norm(Psi(vehicle.Posit,nextLine.Center)
                  - vehicle.Theta));

      if ((lambdaOld != INF) && (lambdaNew*lambdaOld <= 0.0)
      && ((currentPath.config.Kappa == 0.0)
          || (nextLine.config.Kappa == 0.0)
          || (distance(vehicle.Posit, nextLine.Center) <=
              fabs(nextLine.Radius) * 0.9)
          || (angle < HPI*0.9)))

/*  if (testNeutral(nextLine, lambdaNew))*/
    {
      currentPath = GetLine();
      lambdaOld = INF;
    }
      else
    {
      lambdaOld = lambdaNew;
    }
    }
  if (currentPath.Type == TRACKMODE)
    commanded.Linear = computeLinVelocity(actual.Linear,commanded.Linear);
  else
    commanded.Linear = computeLinVelocityS(actual.Linear,commanded.Linear);

  deltaDistance = MOTION_CONTROL_CYCLE * commanded.Linear;

  kappaCommanded = vehicle.Kappa +
    steer(vehicle,currentPath) * deltaDistance;
  commanded.Rotational = kappaCommanded * commanded.Linear;

  return commanded;
}


/*****************************************************************************
  Function  : wallHugRule()
  Purpose   : To determine the linear and rotational velocities needed to put
              or keep Yamabico at the correct offset from the wall.
  Parameters: VELOCITY actual, commanded
```

```
  Returns   : The required linear velocity, rotational velocity
  Comments:   Added by Dan Wells for wall following, May 99.
*****************************************************************************/
/*********************************************/
/****  CODE LISTED IN APPENDIX C OF THESIS  ****/
/*********************************************/

/*****************************************************************************/
double OldLambda()
{
  return lambdaOld;
}

double NewLambda()
{
  return lambdaNew;
}

/*****************************************************************************
  Function  : rotateRule()
  Purpose   : updates the commanded velocity to rotate the robot
  Parameters: VELOCITY actual, commanded
  Returns   : The required linear velocity, rotational velocity
  Comments: last updated 10/25/94 Chien-Liang Chuang
*****************************************************************************/
static  VELOCITY
rotateRule(VELOCITY actual, VELOCITY commanded)
{
   static double goalTheta;
   static int noGoalTheta = TRUE;
   double deltaVel = goalAcc.Rotational * MOTION_CONTROL_CYCLE;
   double rotVel;
   double rotAcc;
   int    Direction;

   rotVel = goalVel.Rotational;
   rotAcc = goalAcc.Rotational;
   if (noGoalTheta)
     {              /* set the goal of rotation */
      goalTheta = vehicle.Theta + currentPath.config.Theta;
      noGoalTheta = FALSE;
     }
   commanded.Linear = 0.0;

   Direction = (currentPath.config.Theta >= 0.0) ? CCW : CW;

            /* First, handle the acceleration */
   if (2.0 * rotAcc * fabs(goalTheta - vehicle.Theta) >
       commanded.Rotational * commanded.Rotational)
     {
      if (Direction == CCW)  /* for CCW rotation */
         commanded.Rotational = Min(commanded.Rotational + deltaVel, rotVel);
      else   /* for clockwise rotation */
         commanded.Rotational = Max(commanded.Rotational - deltaVel, -rotVel);
     }

   else
     {  /* Now, handle the deceleration */
      if (Direction == CCW) /* for CCW rotation */
        {
         if (vehicle.Theta < goalTheta)
            commanded.Rotational = Max(commanded.Rotational - deltaVel, 0.0);
         else
           {
            LEDon(4); /* turn on the LED #4 when finish the rotation */
            currentPath.Type = STOPMODE;
            commanded.Rotational = 0.0;
            noGoalTheta = TRUE;
     }
        }
      else
```

```
        {   /* for CW rotation */
         if (vehicle.Theta > goalTheta)
            commanded.Rotational = Min(commanded.Rotational + deltaVel, 0.0);
         else
           {
            LEDon(4);  /* turn on the LED #4 when finish the rotation */
            currentPath.Type = STOPMODE;
            commanded.Rotational = 0.0;
            noGoalTheta = TRUE;
       }
        }
      }
    return commanded;
}


/*****************************************************************************
  Function:  setLinVelImm()
  Purpose:  sets and updates the linear velocity of the robot
  Parameters: double LinearVelocity
  Returns:  void
  Comments:
*****************************************************************************/
void
setLinVelImm(double linearVelocity)
{
   goalVel.Linear = linearVelocity;
}


/*****************************************************************************
  Function:  setRoVelImm()
  Purpose:  Sets and updates the rotational velocity
  Parameters: double RotationalVelocity
  Returns:  void
  Comments:
*****************************************************************************/
void
setRotVelImm(double RotationalVelocity)
{
   goalVel.Rotational = RotationalVelocity;
}


/*****************************************************************************
  Function:  setLinAccImm()
  Purpose:   Sets and updates the linear acceleration
  Parameters: double LinearAcceleration
  Returns:  void
  Comments:
*****************************************************************************/
void
setLinAccImm(double LinearAcceleration)
{
   goalAcc.Linear = LinearAcceleration;
}


/*****************************************************************************
  Function:  setRotAccImm()
  Purpose:  Sets and updates the rotational acceleration
  Parameters:  double RotationalAcceleration
  Returns: void
  Comments:
*****************************************************************************/
void
setRotAccImm(double RotationalAcceleration)
```

```
{
   goalAcc.Rotational = RotationalAcceleration;
}


/*****************************************************************************
  Function:  setTotalDistanceImm()
  Purpose:  sets the total distance travelled by the robot to the value passed
            as a parameter
  Parameters: double distance
  Returns:  void
  Comments:
*****************************************************************************/
void
setTotalDistanceImm(double distance)
{
   totalDistance = distance;
}



/*****************************************************************************
  Function: getTotalDistanceImm()
  Purpose:  returns the total distance travelled by the robot
  Parameters: void
  Returns: double totalDistance
  Comments:
*****************************************************************************/
double
getTotalDistanceImm(void)
{
   return totalDistance;
}


/*****************************************************************************
  Function: haltMotionImm()
  Purpose:  brings the robot to a rest.  Is different from stop in that it's
            original velocity is saved to be later used by the resume
            command to allow the robot to continue travelling at the same
            speed.
  Parameters: void
  Returns: void
  Comments:
*****************************************************************************/
void
haltMotionImm(void)
{
   if (!Halted)  {
      Halted = TRUE;
      haltedVel.Linear = goalVel.Linear;
      haltedVel.Rotational = goalVel.Rotational;
      WheelsDisable();
   }
}

/*****************************************************************************
  Function: resumeMotionImm()
  Purpose:  Allows the robot to resume the speed it was travelling before the
            haltMotionImm() command was given.
  Parameters: void
  Returns: void
  Comments:
*****************************************************************************/
void
resumeMotionImm(void)
{
   if (Halted)  {
      Halted = FALSE;
      goalVel.Linear = haltedVel.Linear;
      goalVel.Rotational = haltedVel.Rotational;
```

```
      WheelsEnable();
   }
}


/*****************************************************************************
  Function   :  blinkLED()
  Purpose    :  To control the output from the interrupt driven motion control
                system.  LoopTest is assigned zero every second.
  Parameters: void
  Returns    : none
  Comments:
*****************************************************************************/
void
blinkLED(void)
{
   if (LoopTest++ >= ( (int)((1.0/MOTION_CONTROL_CYCLE) - 1)))  {
      changeLEDstate(7);
      LoopTest = 0;
   }
}


/*****************************************************************************
  Function: stopImm()
  Purpose:  updates the goal velocity to zero inorder to stop the robot
  Parameters:  void
  Returns:  void
  Comments: This is the immediate stop command
*****************************************************************************/
void
stopImm(void)
{

   WheelsDisable();
   goalVel.Linear = 0.0;
   goalVel.Rotational = 0.0;

   currentPath.Type = STOPMODE; /*update the path element in motion.c */

}

/*****************************************************************************
  Function   : waitMotionEnd()
  Purpose    : To wait until the motions specified in user program end.
  Parameters: none
  Returns    : void
  Comments: If the conditions are not satisfied, then it stays in busy waiting
*****************************************************************************/
void
waitMotionEnd()
{
 while (!(IsBufferEmpty() && (currentPath.Type == STOPMODE)));
}


/*****************************************************************************
  Function: setSigmaImm()
  Purpose:  sets the size constant which influences the sensitivity of the
                steering function
  Parameters: double sigmaIn
  Returns:  void
  Comments: last updated 10/25/94 Chien-Liang Chuang
*****************************************************************************/
void
setSigmaImm(double sigmaIn)
{
   desiredSigma = sigmaIn;
   return;
}
```

```
/*****************************************************************************
  Function: setSigmaImm()
  Purpose: returns the current value of desiredSigma
  Parameters: void
  Returns: double
  Comments:
*****************************************************************************/
double
sigmaValue()
{
   return desiredSigma;
}



/*****************************************************************************
  Function  : setPathElement()
  Purpose   : To set the current path for the robot
  Parameters: LINE newPath
  Returns   : void
  Comments  : function added by D. Wells 5/5/99 for wall following motion mode
*****************************************************************************/
void
setPathElement(LINE newPath)
{
   currentPath = newPath;
}

/*ADDED used by wheels.c*/
/*****************************************************************************
  Function  : getPathElement()
  Purpose   : retrieves the current path element in the motion control module
  Parameters: void
  Returns   : PATH_ELEMENT
  Comments  :
*****************************************************************************/
LINE
getPathElement(void)
{
   return currentPath;
}


double VelocityLinear()
{

return Commanded.Linear;
}



/*********************************************************************
   FUNCTION : steer(robot,line) PURPOSE : evaluate steering
   function
   *********************************************************************/
double steer(CONFIGURATION robot, LINE line)

{
  double lambda, angle, dist;

  if (line.config.Kappa == 0.0)
      lambda = - line.a * robot.Kappa
           - line.b * norm(robot.Theta - line.config.Theta)
           - line.c *(-(robot.Posit.X - line.config.Posit.X) * sin(line.config.Theta)
                      +(robot.Posit.Y - line.config.Posit.Y) *
cos(line.config.Theta));
  else
    {
      angle = Psi(robot.Posit, line.Center);
      dist = distance(line.Center, robot.Posit);
      if (line.config.Kappa > 0.0)
```

```
    {
    lambda = - line.a * (robot.Kappa-line.config.Kappa)
                - line.b * norm(robot.Theta-(angle-HPI))
             - line.c * (line.Radius - dist);
    }
      else
    lambda = - line.a * (robot.Kappa-line.config.Kappa)
                - line.b * norm(robot.Theta-(angle+HPI))
             - line.c * (line.Radius + dist);
    }
      return lambda;
}

/* This function is grabbing the currentPath before the turn.
We will make more tests later.*/

double waitOnTrack(CONFIGURATION path)
{
  double lambda = 20.0;
  CONFIGURATION robot1;

  double kk;
  double k, kk2;
  LINE pathElement;
  double sigma = sigmaValue();

  pathElement.config = path;
  kk = path.Kappa ;
  kk2 = kk * kk;
  k = 1.0/sigma ;
  pathElement.a = 3.0*k ;
  pathElement.b = 3.0*k*k - kk2 ;
  pathElement.c = k*k*k - 3.0*k*kk2 ;

  do {
    robot1 = getRobotConfig();
    lambda = fabs(pathElement.a * robot1.Kappa)
      + fabs(pathElement.b *
         norm(robot1.Theta - pathElement.config.Theta))
      + fabs(pathElement.c *
         (-(robot1.Posit.X - pathElement.config.Posit.X)
          * sin(pathElement.config.Theta)
          + (robot1.Posit.Y - pathElement.config.Posit.Y)
          * cos(pathElement.config.Theta)));
  } while(lambda > 0.05);
  return lambda;
}



/*****************************************************************************
  Function:  steerByWall(CONFIGURATION, CONFIGURATION, double)
  Purpose:  Computes and returns the steering function value for the
            linear fitted (with decay) wall segment
            tracking based on the current state of the robot
  Parameters: CONFIGURATION robot -- current robot config
              CONFIGURATION curWall -- current linear fitted data for wall
              double offset -- the amount of desired offset from the wall
                               (+ for right wall, - for left wall)
  Returns:  double -- The wall following steering function's value
  Comments: Added by D Wells  4-12-99
*****************************************************************************/
double
steerByWall(CONFIGURATION robot, CONFIGURATION curWall,
            double offset)
{
   double J, deltaTheta, deltaD;
   double k=(1.0/sigmaValue());
   double a, b, c;

   a = 3.0*k;
```

```
    b = 3.0*k*k;
    c = k*k*k;

    deltaTheta = robot.Theta - curWall.Theta;

    deltaD = signedDist(robot.Posit, curWall) - offset;
    J = -a*robot.Kappa
        -b*norm(deltaTheta)
        -c*deltaD;

    return(J);

}



/*****************************************************************************
  File name:  queue.c
*****************************************************************************/
/*
     This file implements the (sequential) instruction buffer. All data
is maintained locally, so all access is done through function calls.

Tue  01-25-94  Frank Kelbe - rewrote entire buffer to redefine the interface
       and access
Wed  01-26-94  removed all references to no_o_paths and skip_flag. "Skipping"
       has now been redefined to mean "flush the buffer as soon as a new
       instruction is ready to be added", and now becomes a pseudo-immediate
       command.
Sun  05-15-94  Cleaned up for inclusion in mml11. Peek... function will need
       to be rewritten when the leaving point calculations are added in.
Fri  07-22-94  FK - modified for mml11 and new leaving point calculations.
       Need to be able to always find the next path segment instruction once
       a new instruction is processed.
Mon  08-22-94  FK - fixed routines in ProcessInstruction for tracking the
       last path segment
Wed  08-24-94  FK - fixed NextSegment calculations. Now NextSegment is only
       looked for when the current instruction catches up.

*/

#include "definitions.h"
#include "system.h"
#include "memory.h"
#include "queue.h"


/*** Local types ***/

typedef struct inst_element  {
    LINE        path;
    ProcessFunctionPtr  Process;    /* the callback function pointer */
}  Instruction;


/*** Local variables ***/
static   int            FlushBufferFlag;
static   int            LineCount;
static   LINE           Buffer[INST_MAX];
static   LINE           *GetPtr,
                        *PutPtr,
                        *HeadPtr,
                        *LastPtr,
                        *NextSegment;


/***  Local Prototypes  ***/
```

```
static void    findNextSegment(void);
static void    IncPtr(LINE **);


/***  Code  ***/

/*****************************************************************************
PURPOSE:           initialize the instruction buffer
PARAMETERS: none
RETURNS:           void
COMMENTS:  Wed  01-19-94  Frank E. Kelbe
   This should only be called one time for initialization
*****************************************************************************/

void
InitQueue(void)
{
   HeadPtr = &Buffer[0];
   LastPtr = &Buffer[INST_MAX-1];

   FlushBuffer();
}



/*****************************************************************************
PURPOSE:    Unconditionally flush the instruction buffer
PARAMETERS: none
RETURNS:           void
COMMENTS:  Wed  01-19-94  Frank E. Kelbe
*****************************************************************************/

void
FlushBuffer(void)
{
   PutPtr = GetPtr = HeadPtr;  /* reset to the start of the buffer */
   LineCount = 0;
   FlushBufferFlag = 0;
   NextSegment = NULL;
}



/*****************************************************************************
PURPOSE:           Used to check the status of the buffer
PARAMETERS: none
RETURNS:           1 if the buffer is empty, 0 otherwise
COMMENTS:  Wed  01-19-94  Frank E. Kelbe
*****************************************************************************/

int
IsBufferEmpty(void)
{
   return (PutPtr == GetPtr);
}

/*****************************************************************************
PURPOSE:    Effectively causes the buffer to be flushed once a new
            instruction arrives
PARAMETERS: none
RETURNS:           void
COMMENTS:   Wed  01-26-94  Frank Kelbe    replaces skip()
*****************************************************************************/
/* allows the queue to remember to flush the buffer as soon as the next
   instruction is received and ready to be added */
void
SkipAllLines(void)
{
   FlushBufferFlag = 1;
}
```

```
/*****************************************************************************
PURPOSE:    Flushes the buffer, then adds the next instruction
PARAMETERS: none
RETURNS:          void
COMMENTS:   Wed  01-26-94  Frank Kelbe
*****************************************************************************/
void
RestartLine(LINE path, ProcessFunctionPtr Process)
{
   FlushBufferFlag = 1;
   AddLine(path);
}


/*****************************************************************************
PURPOSE:    see the next segment to be executed.
PARAMETERS: a pointer to the PATH_ELEMENT to be filled in
RETURNS:    a 1 if there is a path, otherwise a 0
COMMENTS:   Fri  07-22-94  Frank Kelbe
   revised Sun  08-21-94
*****************************************************************************/
int
PeekNextSegment(LINE *path)
{
   if (NextSegment)  {
      *path = *NextSegment;
      return 1;
   }
   else
      return 0;
}


/*****************************************************************************
PURPOSE:          set instruction onto the buffer
PARAMETERS: path of type PATH ELEMENT
RETURNS:          void
COMMENTS:   Wed  01-19-94 Frank Kelbe
*****************************************************************************/
void
AddLine(LINE path)
{
  /*   while (LineCount >= INST_MAX)*/  /* wait for room in the buffer */
    /*  ;      NULL statement */

   if (FlushBufferFlag)     /* flush the buffer now that we have a new */
      FlushBuffer();        /* instruction to add */

   /* DisableInterrupts();*/

   *PutPtr = path;        /* save the path and callback function */

   IncPtr(&PutPtr);
 /*   LineCount++;*/

  /*  EnableInterrupts();*/

}


/*****************************************************************************
PURPOSE: GetLine
PARAMETERS: none
RETURNS:          void
COMMENTS:  May-97
```

```
*****************************************************************/

LINE
GetLine(void)
{

   LINE new;

   new = *GetPtr;
   IncPtr(&GetPtr);

   return new;
}/*****************************************************************************
PURPOSE: ReadLine
PARAMETERS: none
RETURNS:          void
COMMENTS:  May-97
*****************************************************************/

LINE
ReadNextLine(void)
{

   LINE new;

   new = *GetPtr;

   return new;
}



/*****************************************************************************
PURPOSE:          processes the next sequential instruction from
                                        the instruction buffer.
PARAMETERS: none
RETURNS:          void
COMMENTS:  Thu  01-20-94  Frank Kelbe
*****************************************************************/
/*
     All of the callback functions return an int, which mimics the
     use of flg_move in the old code. If flg_move was set ON, the
     functions return 1, otherwise they return 0. A flag move of
     OFF (return of 0) indicates to this function to keep on
     processing instructions. If an ON is returned, processing
     stops
*/

void
ProcessLine(void)
{
   int            StopProcessing = FALSE;
   LINE    *last;

   while (!IsBufferEmpty() && !StopProcessing)
   {
      last = GetPtr;
      /* perform the callback passing a copy of the data
         to the processing function

      StopProcessing = (*(GetPtr->Process))(GetPtr->path);*/
      IncPtr(&GetPtr);
      LineCount--;
   }

   if (last == NextSegment || !NextSegment)
      /* Set the file variable to the next segment in the queue */
   findNextSegment();
}
```

```
/*****************************
    Private Functions
*****************************/

static void
findNextSegment(void)
{
   LINE    *ptr;

   NextSegment = NULL;      /* assume we can't find another path */

   /* No need to look for the next segment if there are no instructions */
   if ((LineCount <= 0) || IsBufferEmpty())
      return;

   ptr = GetPtr;        /* start looking at the next instruction */

      /* keep looking until there are no more instructions. The checking
         is done for all elements in the queue, so the last one found is
         the "next segment".
      */
   while (ptr != PutPtr)  {

   /* !!! This check may have to modified depending on how a
      "next segment" is defined */

      if ( ptr && (
            (ptr->Type == TRACKMODE)
         ))
      {
         NextSegment = ptr;
         break;
      }
      IncPtr(&ptr);
   }
}


static void
IncPtr(LINE **ptr)
{
   ++*ptr;
   if (*ptr > LastPtr)
      *ptr = HeadPtr;
}



/*****************************************************************************
  File name:  seqcmd.c
  Descriptions:  collection of all of the sequential commands that are
                 available to Yamabico
  Revision history:
*****************************************************************************/

#include "definitions.h"
#include "queue.h"
#include "seqcmd.h"
#include "motion.h"
#include "time.h"
#include "stdiosys.h"
#include <math.h>
#include "geometry.h"  /* Added: D Wells 4-13-99 */
#include "sonar.h"  /* Added: D Wells 4-22-99 */
#include "sonarcard.h" /* Added: D Wells 4-19-99 */
#include "sonarmath.h" /* Added: D Wells 4-19-99 */

  /* local variables */
static MODE EndStatusInUser; /* used to record the end status of an instruction
```

```
                                when it is added to buffer in user program */

void
InitSeqcmd(void)
{
    EndStatusInUser = STOPMODE;
}


void
track(CONFIGURATION lineConfig)

{
  double kk;
  double k, kk2;
  LINE pathElement;
  double sigma = sigmaValue();

  pathElement.Type = TRACKMODE;
  pathElement.config = lineConfig;
  kk = lineConfig.Kappa ;

  kk2 = kk * kk;
  k = 1.0/sigma ;
  pathElement.a = 3.0*k ;
  pathElement.b = 3.0*k*k - kk2 ;
  pathElement.c = k*k*k - 3.0*k*kk2 ;
  if (kk == 0.0)
    {
      pathElement.Radius = 0.0;
      pathElement.Center.X = 0.0;
      pathElement.Center.Y = 0.0;
    }
  else
    {
      pathElement.Radius = 1.0/kk;
      pathElement.Center.X = lineConfig.Posit.X  -
    pathElement.Radius * sin(lineConfig.Theta);
      pathElement.Center.Y = lineConfig.Posit.Y +
    pathElement.Radius * cos(lineConfig.Theta);
    }
  AddLine(pathElement);
  return;
}

void
trackS(CONFIGURATION lineConfig)

{
  double kk;
  double k, kk2;
  LINE pathElement;
  double sigma = sigmaValue();

  pathElement.Type = TRACKSMODE;
  pathElement.config = lineConfig;
  kk = lineConfig.Kappa ;

  kk2 = kk * kk;
  k = 1.0/sigma ;
  pathElement.a = 3.0*k ;
  pathElement.b = 3.0*k*k - kk2 ;
  pathElement.c = k*k*k - 3.0*k*kk2 ;
  if (kk == 0.0)
    {
      pathElement.Radius = 0.0;
      pathElement.Center.X = 0.0;
      pathElement.Center.Y = 0.0;
    }
  else
```

```
    {
      pathElement.Radius = 1.0/kk;
      pathElement.Center.X = lineConfig.Posit.X  -
      pathElement.Radius * sin(lineConfig.Theta);
      pathElement.Center.Y = lineConfig.Posit.Y +
    pathElement.Radius * cos(lineConfig.Theta);
    }
  AddLine(pathElement);
  return;
}

/************
void
trackWall(int sonarNum)

{
  double k;
  LINE pathElement;
  double sigma = sigmaValue();
  double offset;

  /* sonarNum determines side of wall following          */
  /* Constants defined in sonar.h                        */
  /* S090 - wall on right side, S270 - wall on left side *
  switch (sonarNum) {
     case S090:
        pathElement.Type = TRACKRWALLMODE;
        offset = DEFAULT_WALL_OFFSET;
        break;
     case S270:
        pathElement.Type = TRACKLWALLMODE;
        offset = - DEFAULT_WALL_OFFSET;
        break;
     default:
        printf("\nERROR: illegal value for sonarNum in trackWall()");
        exit(1);
        break;
  }

  pathElement.config = defineConfig(0.0, offset, 0.0, 0.0);

  k = 1.0/sigma ;
  pathElement.a = 3.0*k ;
  pathElement.b = 3.0*k*k;
  pathElement.c = k*k*k;
  pathElement.Radius = offset;
  pathElement.Center.X = 0.0;
  pathElement.Center.Y = 0.0;

  AddLine(pathElement);

  EnableSonar(sonarNum);
  EnableLinearFittingDecay(sonarNum);

  return;
}
**********/


/*****************************************************************************
FUNCTION:   Rotate (sequential)
PURPOSE:    Rotate the robot by Theta radians.
            Positive is counterclockwise and negative is clockwise.
*****************************************************************************/
void
Rotate(double Theta)
{
   LINE path;

   if (EndStatusInUser != STOPMODE)
      /* do nothing if it follows a motion without stop*/
```

```
      return;

   path.Type = ROTATEMODE;

      /* here is where the union would be cleaner */
   path.config.Theta = Theta;
   EndStatusInUser = STOPMODE;
   AddLine(path);
}



#ifdef CODE_IS_DONE

/*****************************************************************************
FUNCTION:  set_error sequential
PARAMETERS: code
PURPOSE:           set an error code into the instruction buffer.
RETURNS:           void
CALLED BY: rotate, solve, switch_dir, set_rob
CALLS:                    set_inst();
COMMENTS:  7 Jan 93 - Dave MacPherson
                                          Tue  01-25-94  Frank Kelbe - modified to use
new buffer
TASK: Level 0
*****************************************************************************/

void
set_error(ERROR_CODE code)
{
   PATH_ELEMENT path;

   path.mode = ERRORMODE;
   path.type = code;
   AddLine(path);
}


int
ErrorProcess(PATH_ELEMENT path)
{
   DispError(path.type);
   halt();
   change_status(SERROR);
   return 1;
}



/*************************************************************************
NAME :                    switch_dir
ARGUMENTS : none
FUNCTION : to reverse the heading direction of the robot
COMMENTS:  Tue  01-25-94  Frank Kelbe - modified to use new buffer
*************************************************************************/
void
switch_dir(void)
{
     PATH_ELEMENT path;

     if (seq_status != SSTOP)  {
                  set_error(ECODE2);
                  return;
     }

     path.type = SWITCH;
         path.pathType.mode = STOPMODE;
     AddLine(path);

     nom_p->t = norm(nom_p->t + PI);
```

```
}


/*************************************************************************
NAME :                    set_rob
ARGUMENTS : Configuration to set robot's location to.
FUNCTION : to add set robot sequence to queue
COMMENTS:  Tue  01-25-94  Frank Kelbe - modified to use new buffer
*************************************************************************/
void
set_rob(CONFIGURATION *pst)
{
   PATH_ELEMENT path;

   if(seq_status != SSTOP)  {
      set_error(ECODE2);
      return;
   }

   path.pc = *pst;
   vehicle.x = pst->x;
   vehicle.y = pst->y;
   vehicle.t = pst->t;
   vehicle.k = pst->k;
   nom_p->x = pst->x;
   nom_p->y = pst->y;
   nom_p->t = pst->t;
   nom_p->k = pst->k;
   path.type = SET_ROB;
   path.pathType.mode = STOPMODE;
   AddLine(path);

   last_robot_path_element.pc = *pst;
   last_robot_path_element.type = SET_ROB;
}

#endif



/*************************************************************
 FILENAME:    sonar.h
 PURPOSE :    include file for sonar.
 AUTHOR  :
 DATE    :
 COMMENTS:    cleaned and documented by Mahmoud Wahdan and
              Data structure LINE_SEG is removed and added
              to SEGMENT and SEGMENT changed to SEGMENT_RES &
              m00 is added to the new data structure. CUR_DATA
              is changed to SEGMENT_WORK.
*************************************************************/
#ifndef __SONAR_H
#define __SONAR_H

#include "constants.h"
#include "definitions.h"


#define     NUM_SONARS     16


          /* Sonar locations */
          /* --------------- */

#define S000          0
#define S030          3
#define S060          10
#define S090          7
```

```
#define S120          6
#define S150          9
#define S180          2
#define S210          1
#define S240          8
#define S270          5
#define S300          4
#define S330          11


          /* Types of sonar logging */
          /* ---------------------- */

#define      SONAR_NONE         0x00
#define      SONAR_RAW          0x01
#define      SONAR_GLOBAL       0x02
#define      SONAR_SEGMENT      0x04
#define      SONAR_ALL          0x07


#define      SONAR_CTL          0xfc0083f9

          /* The data structure SONARD */
          /* ------------------------- */

typedef struct  {
   int       fitting;      /* flag to indicate linear fitting request    */
   int       decayFitting; /* flag to indicate linear fitting w/ decay   */
   int       update;       /* flag to indicate presence of new data      */

   POINT     global;       /* global position of sonar return            */
   double    d;            /* range data                                 */
   double    d0;           /* old range data                             */
   POINT     posit;        /* robot's position at time of range          */
   double    t;            /* robot's orientation angle at time of range */

   POINT     SonarPosit;   /* position of sonar from center              */
   double    SonarTheta;   /* angle of sonar from robot center           */

}  SONARD;


          /* defines a basic segment with the start and end points, */
          /* the sonar number associats and the length of segment   */
          /* ------------------------------------------------------ */

          /* modified by Mahmoud Wahdan on 03-06-95 */
          /* -------------------------------------- */

typedef struct  {
   int       SonarNumber;  /* the sonar no associates with a segment     */
   double    m00;          /* # points of least squares LF               */
   POINT     start;        /* segment with the start and end points      */
   POINT     end;          /* headx, heady, tailx, taily                 */
   double    alpha;        /* angle and length of normal from origin     */
   double    r;            /* to the segment                             */
   double    length;       /* length of the segment                      */
}  SEGMENT_RES;



          /* defines a basic segment defined by linear fitting,     */
          /* with a decay factor                                    */
          /* ------------------------------------------------------ */

          /* modified by Dan Wells on 04-15-99      */
          /* -------------------------------------- */

typedef struct  {
   int       SonarNumber;  /* the sonar no associates with a segment     */
   int       usable;       /* flag to indicate at least 2 data points    */
```

```
   double   numPoints;     /* # points of least squares LF w/ decay      */
   double   alpha;         /* angle and length of normal from origin     */
   double   r;             /* to the segment                             */
   double   decayFactor;   /* the decay factor (range: ~0 -> 1)          */
                           /*   1 => no decay, all points equal weight   */
                           /*  ~0 => previous points have no weight      */
   double   m00,           /* moments of linear fitting algorithm        */
            m10,
            m01,
            m11,
            m20,
            m02;
}  SEGMENT_RES_DECAY;

         /* modified by Mark Merrell on 1-11-99 */
         /* ----------------------------------- */

typedef struct {
   int      SonarNumber;  /* the sonar no associates with a segment     */
   double   m00;          /* # points of least squares LF               */
   POINT    start;        /* segment with the start and end points      */
   POINT    end;          /* headx, heady, tailx, taily                 */
   double   alpha;        /* angle and length of normal from origin     */
   double   r;            /* to the segment                             */
   double   length;       /* length of the segment                      */
}  ARRAY_RES;

         /* revised by Y. Kanayama, 07-07-93 */
         /* -------------------------------- */

typedef struct  {
   double       m00,          /* moments of least squares fitting algotithm */
                m10,
                m01,
                m20,
                m11,
                m02;
   SEGMENT_RES  seg;          /* sonar no., startx, starty, endx, endy,     */
                              /* alpha, r, length                           */
}  SEGMENT_WORK;


         /* Global variables */
         /* ---------------- */

extern   SONARD    sonar_table[];
extern   ARRAY_RES SegmentArray[NUM_SONARS][100];

         /* Prototype */
         /* ---------- */

void          InitSonar(void);


         /* Interrupt handler */
         /* ----------------- */

void          SonarSysControl(void);

#endif



/*************************************************************************
 FILENAME      : sonar.c

 PURPOSE       : Provides the global generic sonar functions

 CONTAINS      : InitSonar()
```

```
                SonarSysControl()

 AUTHOR         : Patrick Byrne, Yutaka Kanayama

 DATE           : 20 November 1993

 COMMENTS       : Fri  07-22-94  Updated for Sparc mml11 FEK
                : updated by Khaled morsy 11-22-94
                : cleaned and documented on Mon 3-6-96 by Mahmoud Wahdan
*****************************************************************************/

#include "definitions.h"
#include "memsys.h"
#include "sonar.h"
#include "motion.h"
#include "sonarcard.h"
#include "sonarmath.h"
#include "sonarlog.h"
#include "sparc.h"
#include "time.h"
#include "trace.h"

#ifdef TIMER
#include "trace.h"
#include "clocktick.h"

#define  TIMER_FILE         "timer.log"
#define  TIMER_SIZE         0x10000
#define  TIMER_FREQUENCY    1

IOhandle TimerLog;
#endif


        /* Global variables */
        /* ---------------- */

SONARD   sonar_table[NUM_SONARS];       /* SONARD struc & NUM_SONARS = 16 */
                                        /* are defined in sonar.h         */
 ARRAY_RES SegmentArray[NUM_SONARS][100];

    /* used by ServeSonar */
        /* ------------------ */

static const int group_array[4][4] =   /* array maps sonar no to groups */
{
   {0, 5, 2, 7},
   {3, 4, 1, 6},
   {10, 11, 8, 9},
   {12, 13, 14, 15}
};

/*****************************************************************************
 FUNCTION       : InitSonar()
 PARAMETERS     : None
 PURPOSE        : initializes sonar table and sets up compensation for sonar
                : configurations.
 RETURNS        : void
 CALLS          : InitSonarmath() in sonarmath.h.
                : InitSonarlog() in sonarlog.h.
                : SetSonarParameters() in sonarmath.h.
                : memset() in memsys.h.
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
InitSonar(void)
{
   int    i;
```

```
        /* initialize sonar_table */
        /* ---------------------- */

for (i = 0; i < NUM_SONARS; i++)  /* NUM_SONARS = 16 is defined in sonar.h */
  {
 memset(&sonar_table[i], 0, sizeof(SONARD));
   sonar_table[i].fitting  = 0;
   sonar_table[i].decayFitting  = 0;
   sonar_table[i].update   = 0;
   sonar_table[i].d        = 0.0;
   sonar_table[i].d0       = 0.0;
   sonar_table[i].global.X = 0.0;
   sonar_table[i].global.Y = 0.0;
   sonar_table[i].t        = 0.0;
   sonar_table[i].posit.X  = 0.0;
   sonar_table[i].posit.Y  = 0.0;
  }

 /* set up compensation for new sonar configurations 12/08/94 */
     /* ------------------------------------------------------- */

sonar_table[0].SonarTheta = 0.0;
sonar_table[1].SonarTheta = 5.0*PI/6.0;
sonar_table[2].SonarTheta = PI;
sonar_table[3].SonarTheta = -PI/6.0;
sonar_table[4].SonarTheta = PI/3.0;
sonar_table[5].SonarTheta = PI/2.0;
sonar_table[6].SonarTheta = -2.0*PI/3.0;
sonar_table[7].SonarTheta = -PI/2.0;
sonar_table[8].SonarTheta = 2.0*PI/3.0;
sonar_table[9].SonarTheta = -5.0*PI/6.0;
sonar_table[10].SonarTheta = -PI/3.0;
sonar_table[11].SonarTheta = PI/6.0;
sonar_table[12].SonarTheta = 0.0;
sonar_table[13].SonarTheta = 1.5708;
sonar_table[14].SonarTheta = 4.7124;
sonar_table[15].SonarTheta = 0.0;

sonar_table[0].SonarPosit.X = 23.6;
sonar_table[1].SonarPosit.X = -23.0;
sonar_table[2].SonarPosit.X = -22.6;
sonar_table[3].SonarPosit.X = 24.7;
sonar_table[4].SonarPosit.X = 13.4;
sonar_table[5].SonarPosit.X = 0.0;
sonar_table[6].SonarPosit.X = -12.6;
sonar_table[7].SonarPosit.X = 0.0;
sonar_table[8].SonarPosit.X = -13.4;
sonar_table[9].SonarPosit.X = -23.5;
sonar_table[10].SonarPosit.X = 12.1;
sonar_table[11].SonarPosit.X = 25.2;
sonar_table[12].SonarPosit.X = 0.0;
sonar_table[13].SonarPosit.X = 1.5708;
sonar_table[14].SonarPosit.X = 4.7124;
sonar_table[15].SonarPosit.X = 0.0;

sonar_table[0].SonarPosit.Y = -0.5;
sonar_table[1].SonarPosit.Y = 13.1;
sonar_table[2].SonarPosit.Y = -1.0;
sonar_table[3].SonarPosit.Y = -14.6;
sonar_table[4].SonarPosit.Y =  21.3;
sonar_table[5].SonarPosit.Y = 20.6;
sonar_table[6].SonarPosit.Y = -21.3;
sonar_table[7].SonarPosit.Y = -20.5;
sonar_table[8].SonarPosit.Y = 21.3;
sonar_table[9].SonarPosit.Y = -14.9;
sonar_table[10].SonarPosit.Y = -21.3;
sonar_table[11].SonarPosit.Y = 14.1;
sonar_table[12].SonarPosit.Y = 0.0;
sonar_table[13].SonarPosit.Y = 21.5;
sonar_table[14].SonarPosit.Y = 21.5;
sonar_table[15].SonarPosit.Y = 0.0;
```

```
        /* initialize the sonar components */
        /* ------------------------------ */

   InitSonarmath();                /* it is defined in sonarmath.h */
   InitSonarlog();                 /* it is defined in sonarlog.h  */

   SetSonarParameters(0.02, 5.0);  /* it is defined in sonarmath.h */

#ifdef TIMER
   TimerLog = IOopen(TIMER_FILE, TIMER_SIZE, TIMER_FREQUENCY);
#endif


}

/*****************************************************************************
 FUNCTION       : SonarSysControl()
 PARAMETERS     : None
 PURPOSE        : It is the ``central command'' for the control of all sonar related
                : functions. It is linked with the ih_sonar routine and loads sonar
data
                : to the sonar_table to determine which activities the user wishes to
                : take place, and calls the appropriate functions. This procedure is
                : invoked approximately every thirty msec by an interrupt from the
sonar
                : control board.
 RETURNS        : None
 CALLS          : CalculateGlobal() in sonarmath.h.
                : LinearFitting() in sonarmath.h.
                : LinearFittingDecay() in sonarmath.h.
                : LogSonarData() in sonarlog .h.
                : changeLEDstate() in system.h.
                : getRobotConfig() in motion.h.
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
SonarSysControl(void)
{

#define OVERFLOWMASK    0x8000          /* overflow bit is bit 15 */
#define GROUP_MASK      0x18
#define SONAR_DATA0     0xfc0083f0
#define SONAR_DATA1     0xfc0083f2
#define SONAR_DATA2     0xfc0083f4
#define SONAR_DATA3     0xfc0083f6

   static int        cnt = 0;
   int               n;
   int               i;
   int               data[4];
   int               group;
/*   double             t;      */        /* uses only in TIMER */
   CONFIGURATION     current;            /* is defined in definition.h */

#ifdef TIMER
   int   tick = getCount();
   IOprintf(TimerLog, "%f ", tick / 250.0);
#endif

   if (++cnt > 10)                       /* blink the #1 LED */
     {
      cnt = 0;
      changeLEDstate(1);                 /* is defined in system.h */
     }

   current = getRobotConfig();           /* getRobotConfig() is defined */
                                         /* motion.h */
```

```
   group   = ((*(BYTE*)SONAR_CTL & GROUP_MASK) >> 3);

   data[0] = *(WORD*)SONAR_DATA0;
   data[1] = *(WORD*)SONAR_DATA1;
   data[2] = *(WORD*)SONAR_DATA2;
   data[3] = *(WORD*)SONAR_DATA3;

   for (i = 0; i < 4; i++)
     {
      n = group_array[group][i];

      sonar_table[n].posit.X = current.Posit.X;
      sonar_table[n].posit.Y = current.Posit.Y;
      sonar_table[n].t = current.Theta;
      sonar_table[n].d0      = sonar_table[n].d;

      if (data[i] & OVERFLOWMASK)
         sonar_table[n].d = INFINITY;
      else
        {
         data[i] &= 0xfff;                /* only first 12 bits are data, */
                                          /* so mask the data */

         sonar_table[n].d = data[i] * 0.1029;
        }

      sonar_table[n].update = 1;

      CalculateGlobal(n);                 /* is defined in sonarmath.c */

      if (sonar_table[n].fitting == 1)
         LinearFitting(n);                /* is defined in sonarmath.c */

      if (sonar_table[n].decayFitting == 1)
         LinearFittingDecay(n);           /* is defined in sonarmath.c */

   /* LogSonarData(n); */          /* this function is moved to */
                                   /* CalculateGlobal() in sonarmath.c */
                                   /* because interrupt affects it & some */
                                   /* logging numeric digits becomes character */
                                   /* done by Mahmoud wahdan Aug. 16 95 */
    }

#ifdef TIMER
   t = (tick - getCount())/250.0;
   if (t <  0.0)
      t = t + 10.0;
   IOprintf(TimerLog, "%f \n", t );
#endif

 }



/*************************************************************************
 FILENAME       : sonarcard.c

 PURPOSE        : contains functions for sonar.

 CONTAINS       : InitSonarcard()
                  EnableSonar()
                  DisableSonar()
                  DisableAllSonar()
                  EnableSonarInterrupts()
                  DisableSonarInterrupts()

 AUTHOR         : Patrick Byrne

 DATE           : 20 November 1993
```

```
 COMMENTS      : Fri  07-22-94  Updated for Sparc mmll1 FEK
               : cleaned and documented on Mon 3-6-96 by Mahmoud Wahdan
*************************************************************************/

#include "definitions.h"
#include "sonar.h"
#include "system.h"
#include "time.h"
#include "sonarcard.h"
#include <io_defs.h>
#include <reg.h>
#include <traptypes.h>

               /******  Local variables  ******/
               /******  ---------------  ******/

static char  *command_ptr;
static char  *BIM_ptr;
static int   enabled_sonars[NUM_SONARS];/* NUM_SONAR = 16 is defined in sonar.h */
static char  enabled;


#define   SONAR_VECTOR       0xa2

#define   BIM0_CTL0          0xfc0080c1      /* BIM0 control register */

#define   BIM0_VECT0         0xfc0080c9      /* BIM0 vector register */

#define   IRQ2               0x12

#define   SONAR_CTL          0xfc0083f9  /* sonar board command/status register */

#define      IRQ2_REG           0xfffc000b


int
SonarSysControlWrapper(TrapType trap, struct reg_save *reg, int vector)
{

 SonarSysControl();
 return IO_INTR_EXPECTED;
}


               /****** Source Code ******/
               /****** ----------- ******/

/*****************************************************************************
 FUNCTION      : InitSonarcard()
 PARAMETERS    : None
 PURPOSE       : It initializes enabled_sonars list and sonar board command/status
register.
 RETURNS       : void
 CALLS         : DisableInterrupts() in system.h.
               : EnableInterrupts() in system.h.
               : mk_handler() in sparc.h.
 CALLED BY     : None
 COMMENTS      : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
InitSonarcard(void)
{
   int   i;
   #define SonarIntVector          0xa2

   command_ptr = (char *)SONAR_CTL;
   BIM_ptr     = (char *)BIM0_CTL0;
   enabled = 0;
```

```
   for (i=0; i<NUM_SONARS; i++)
      enabled_sonars[i] = 0;                    /* Initialization */

   DisableInterrupts();                         /* defined in system.h */

   /* This function is not defined */
   mk_handler(SonarIntVector, SonarSysControlWrapper); /* defined in sparc.h */

   *(BYTE*)BIM0_VECT0 = SONAR_VECTOR;
   *(BYTE*)BIM0_CTL0 = IRQ2;                    /* enable interrupts and select IRQ2
*/
   *(BYTE*)SONAR_CTL = 8;                       /* zeroes the command register */
   *(BYTE*)IRQ2_REG = 0x7a;

   EnableInterrupts();                          /* defined in system.h */
}

/*****************************************************************************
 FUNCTION       : EnableSonar(SonarNumber)
 PARAMETERS     : int SonarNumber
 PURPOSE        : It enables the sonar group that contains sonar SonarNumber, which
causes
                : all the sonars in that group to echo-range and write data to the data
                : registers on the sonar control board. Makes the SonarNumber'th
position
                : of the enabled_sonars array to track which sonars are enabled.
 RETURNS        : void
 CALLS          : DisableSonarInterrupts() in sonarcard.h.
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
EnableSonar(int SonarNumber)
{

   DisableSonarInterrupts();            /* defined in sonarcard.h */
   enabled_sonars[SonarNumber] = 1;
   switch (SonarNumber) {
      case 0:
      case 2:
      case 5:
      case 7:
         enabled |= 0x01;
         break;
      case 1:
      case 3:
      case 4:
      case 6:
         enabled |= 0x02;
         break;
      case 8:
      case 9:
      case 10:
      case 11:
         enabled |= 0x04;
         break;
      case 12:
      case 13:
      case 14:
      case 15:
         enabled |= 0x08;
         break;
   } /* switch */

   EnableSonarInterrupts();             /* defined in sonarcard.h */
   *command_ptr = enabled;
}

/*****************************************************************************
 FUNCTION       : DisableSonar(SonarNumber)
```

```
 PARAMETERS     : int SonarNumber
 PURPOSE        : It removes the sonar SonarNumber from the enabled_sonars list. If
sonar
                : SonarNumber is the only enabled sonar from it's group, then the group
                : is disabled as well and will stop echo ranging. This has benefit of
                : shortening the ping interval for groups that remain enabled.
 RETURNS        : void
 CALLS          : DisableSonarInterrupts() in sonarcard.h
                : EnableSonarInterrupts() in sonarcard.h.
 CALLED BY      : DisableAllSonar() in sonarcard.h.
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
DisableSonar(int SonarNumber)
{
   int   c;

   DisableSonarInterrupts();            /* defined in sonarcard.h */
   enabled_sonars[SonarNumber] = 0;

   switch (SonarNumber)
   {
      case 0:
      case 2:
      case 5:
      case 7:
         c = enabled_sonars[0] + enabled_sonars[2] +
             enabled_sonars[5] + enabled_sonars[7];
         if (c == 0)
            enabled &= 0xfe;
         break;
      case 1:
      case 3:
      case 4:
      case 6:
         c = enabled_sonars[1] + enabled_sonars[3] +
             enabled_sonars[4] + enabled_sonars[6];
         if (c == 0)
            enabled &= 0xfd;
         break;
      case 8:
      case 9:
      case 10:
      case 11:
         c = enabled_sonars[8] + enabled_sonars[9] +
             enabled_sonars[10] + enabled_sonars[11];
         if (c == 0)
            enabled &= 0xfb;
         break;
      case 12:
      case 13:
      case 14:
      case 15:
         c = enabled_sonars[12] + enabled_sonars[13] +
             enabled_sonars[14] + enabled_sonars[15];
         if (c == 0)
            enabled &= 0xf7;
         break;
   } /* switch */

   *command_ptr = enabled;
   EnableSonarInterrupts();             /* defined in sonarcard.h */
}

/*****************************************************************************
 FUNCTION       : DisableAllSonar()
 PARAMETERS     : None
 PURPOSE        : removes the all sonars from the enabled_sonars list.
 RETURNS        : void
 CALLS          : DisableSonar() in sonarcard.h.
```

```
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
DisableAllSonar(void)
{
   int   i;

   for (i=0; i < 12; i++)
     DisableSonar(i);                    /* defined in sonarcard.h */
}

/*****************************************************************************
 FUNCTION       : EnableSonarInterrupts()
 PARAMETERS     : None
 PURPOSE        : places sonar control board in interrupt driven mode.
 RETURNS        : void
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
EnableSonarInterrupts(void)
{
   *BIM_ptr |= 0x10;                    /* cordeiro code has *BIM_ptr = 0x12; */
}

/*****************************************************************************
 FUNCTION       : DisableSonarInterrupts()
 PARAMETERS     : None
 PURPOSE        : stops interrupt generation by the sonar control board. A flag is set
                : in the status register when data is ready, and it is the user's
                : responsibility to poll the sonar system for the flag.
 RETURNS        : void
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
DisableSonarInterrupts(void)
{
   *BIM_ptr &= 0xef;                    /* cordeiro code has *BIM_ptr = 0; */
}



/*********************************************************************
 FILENAME       : sonarmath.c

 PURPOSE        : Provides the main sonar functions

 CONTAINS       : InitSonarmath()
                  Sonar()
                  SonarNew()
                  Global()
                  CalculateGlobal()
                  SetSonarParameters()
                  EnableLinearFitting()
                  DisableLinearFitting()
                  LinearFitting()
                  AddToSegment()       * this function is local in sonarmath.c *
                  GenerateSegment()
                  EndSegment()
                  BuildList()          * this function is local in sonarmath.c *
                  ResetMoments()       * this function is local in sonarmath.c *
                  GetLastSegment()
                  GetCurrentSegment()
```

```
                GetSegmentConfig()
                WaitSegment()
                CorrectOdometryError()

               ** Linear Fitting with Decay      **
               ** Added by Dan Wells on 04-15-99 **
                ResetMomentsDecay()
                EnableLinearFittingDecay()
                DisableLinearFittingDecay()
                LinearFittingDecay()
                AddToSegmentDecay()
                GetSegmentDecay()
                SetDecayFactor()


 AUTHOR         : Patrick Byrne

 DATE           : 20 November 1993

 COMMENTS       : Fri  07-22-94  Updated for Sparc mml11 FEK
                : cleaned and documented on Mon 3-6-96 by Mahmoud Wahdan
                : and data structure LINE-SEG is removed and added to SEGMENT_RES.
                : and CUR_DATA is changed to SEGMENT_WORK.
                : Refer to this data structure in sonar.h
*************************************************************************/

#include "definitions.h"
#include "stdiosys.h"
#include "sonar.h"
#include "math.h"
#include "motion.h"
#include "memsys.h"
#include "sonarlog.h"
#include "sonarmath.h"
#include "trace.h"
#include "geometry.h"

/*ADDED*/
#include "sparc.h"

#define SIZE   30

/* #define DEBUG */
/*
 * the following constants are used in logging functions for debugging
 * purposes
*/

#ifdef DEBUG
#define  DEBUG_FILE        "debug.log"
#define  DEBUG_SIZE        0x10000
#define  DEBUG_FREQUENCY   1

IOhandle        debugIO;
#endif


/* #define LOCALIZATION */
/*
 * the following constants are used in logging functions for localization purpose */


#ifdef LOCALIZATION
  #define LOCALIZATION_FILE           "localization.log"
  #define  LOCALIZATIONDEBUG_SIZE     0x10000
  #define  LOCALIZATION_FREQUENCY     1

  IOhandle        localizationIO;
#endif
```

```
            /* Local variables */
            /* --------------- */

static double         C1, C2;

static SEGMENT_RES    segstruct ;                    /* temporary storage for
                                                        EndSegment func.*/

/*static int          SegListHead[NUM_SONARS];*/     /* points to oldest segment
                                                        array element  */

static int            SegListTail[NUM_SONARS];       /* points to newest segment
                                                        array element */

static SEGMENT_RES    seg_list[NUM_SONARS][SIZE];    /* segments for working
                                                        memory */

static SEGMENT_WORK   segment_data[NUM_SONARS];      /* interim data for all
                                                        sixteen sonars CUR_DATA
                                                        is changed to SEGMENT_WORK */

static SEGMENT_RES_DECAY  seg_data_decay[NUM_SONARS]; /* data for sonars using
                                                        linear fitting with
                                                        decay factor */

static SEGMENT_RES    *seg_end_config;               /* temporary storage for
                                                        segment being built is
                                                        completed */

            /* Local Prototypes */
            /* ---------------- */

void     AddToSegment(int, POINT);
void     ResetMoments(int);
void     BuildList(SEGMENT_RES*, int);
void     AddToSegmentDecay(int, POINT);

                           /* Source Code */
                           /* ----------- */


/*****************************************************************************
 FUNCTION      : GetSegment(int SonarNumber, int SegmentNumber)
 PARAMETERS    : int SonarNumber, int SegmentNumber
 PURPOSE       : Gets a particular segment
 RETURNS       : SEGMENT_RES
 CALLS         : None
 CALLED BY     : user.c
 COMMENTS      : Created by Mark Merrell 20 May 1998
*****************************************************************************/

SEGMENT_RES *
GetSegment(int SonarNumber, int SegmentNumber)
{
   SEGMENT_RES Segment;

   Segment = seg_list[SonarNumber][SegmentNumber];

   return & Segment;


}


/*****************************************************************************
 FUNCTION      : GetNumPoints(int SonarNumber)
 PARAMETERS    : int SonarNumber
 PURPOSE       : Gets a number of points in the current segment
```

```
 RETURNS       : int
 CALLS         : None
 CALLED BY     : user.c
 COMMENTS      : Created by Mark Merrell 14 July 1998
*****************************************************************************/

int
GetNumPoints(int SonarNumber)
{

   return segment_data[SonarNumber].m00;

}


/*****************************************************************************
 FUNCTION      : GetNumSegments(int SonarNumber)
 PARAMETERS    : int SonarNumber
 PURPOSE       : Returns the number of Segments
 RETURNS       : int
 CALLS         : None
 CALLED BY     : user.c
 COMMENTS      : Created by Mark Merrell 20 May 1998
*****************************************************************************/

int
GetNumSegments(int SonarNumber)
{

   return SegListTail[SonarNumber];
}

/*****************************************************************************
 FUNCTION      : InitSonarmath()
 PARAMETERS    : None
 PURPOSE       : It resets the accumulative values m00, m10, m01, m20, m11,
                 m02 to zero.
 RETURNS       : void
 CALLS         : memset() in memsys.h.
               : ResetMoments() in sonarmath.c  *Local Prototype's in sonarmath.c*
               : ResetMomentsDecay() in sonarmath.c  *Local Prototype's in
sonarmath.c*
 CALLED BY     : InitSonar() in sonar.h.
 COMMENTS      : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
InitSonarmath(void)
{
   int  i;

   for (i = 0; i < NUM_SONARS; i++)      /* NUM_SONARS 16 is defined in sonar.h */
   {
      ResetMoments(i);                     /* defined in sonarmath.c */
      ResetMomentsDecay(i);                /* defined in sonarmath.c */

      SegListTail[i] = -1;

      segment_data[i].seg.start.X = 0.0;
      segment_data[i].seg.start.Y = 0.0;
      segment_data[i].seg.end.X   = 0.0;
      segment_data[i].seg.end.Y   = 0.0;
      segment_data[i].seg.alpha   = 0.0;
      segment_data[i].seg.r       = 0.0;
      segment_data[i].seg.length  = 0.0;
      segment_data[i].seg.m00     = 0.0;
      segment_data[i].seg.SonarNumber = i;

    }
```

```
}

/******************** Source Code For Sonar ***************************/
/*                    ======================                          */


/*****************************************************************************
 FUNCTION       : Sonar(int SonarNumber)
 PARAMETERS     : int SonarNumber
 PURPOSE        : It returns the distance (in centimeters) sensed by the
                : SonarNumber ultrasonic sensor.
                : If no echo is received,
                :          then INFINITY (1.0$e$6) is returned.
                : If the distance is less than 10 cm,
                :          then a 0 is returned.
 RETURNS        : the distance (in centimeters) sensed by the SonarNumber
                : ultrasonic sensor.
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

double
Sonar(int SonarNumber)
{
   sonar_table[SonarNumber].update = 0;
   return sonar_table[SonarNumber].d;  /* sonar_table is global variable which
                                          is defined in sonar.c
                                          the data structure of sonar_table is
                                          SONARD which is defined in sonar.h */
}

/*****************************************************************************
 FUNCTION       : SonarNew(int SonarNumber)
 PARAMETERS     : int SonarNumber
 PURPOSE        : waits in a loop until new data ia available for sonar SonarNumber.
 RETURNS        : the range value for sonar number.
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

double
SonarNew(int SonarNumber)
{
   while (sonar_table[SonarNumber].update == 0)

        ;                               /* NULL statement */

   sonar_table[SonarNumber].update = 0;

   return sonar_table[SonarNumber].d;
}

/*****************************************************************************
 FUNCTION       : Global(int SonarNumber)
 PARAMETERS     : int SonarNumber
 PURPOSE        : It returns a structure of type POINT containing the global x
                : and y coordinates of the position of the last sonar return.
 RETURNS        : the global x and y coordinates of the position of the last
                : sonar return.
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

POINT
Global(int SonarNumber)
{
   return  sonar_table[SonarNumber].global;
}
```

```
/*****************************************************************************
 FUNCTION       : CalculateGlobal(int SonarNumber)
 PARAMETERS     : int SonarNumber
 PURPOSE        : It calculates the global x and y coordinates for the range value
                : and robot configuration in the sonar table. The results are
                : stored in the sonar table.
 RETURNS        : None
 CALLS          : getRobotConfig()      in motion.h.
                : LogSonarData()        in sonarlog.h
                : DisableInterrupts()   in sparc.c
                : EnableInterrupts()    in sparc.c
 CALLED BY      : SonarSysControl()     in sonar.h.
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
                : DisableInterrupts(), EnableInterrupts() & LogSonarData() added by
                : Mahmoud wahdan on Aug 16,95 to solve the problems of logging data
                : (some numeric digits becomes characters and some numbers becomes
                : very big or very small (wrong calculation).
*****************************************************************************/

void
CalculateGlobal(int SonarNumber)
{
   double  lx, ly, lt, range;            /* describes the sensor's configuration
                                            in the global coordinate system */
   double  sonarX, sonarY, sonarTheta;   /* describes the sensor's configuration
                                            in the local coordinate system */
   double  robotX, robotY, robotTheta;   /* describes the robot current configuration
                                            in the global coordinate system */

   DisableInterrupts();

   range = sonar_table[SonarNumber].d;

   if (range >= INFINITY0)
   {
      sonar_table[SonarNumber].global.X = INFINITY; /* sonar_table is global variable
*/
      sonar_table[SonarNumber].global.Y = INFINITY; /* which is defined in sonar.c */
   }                                                /* the structure of sonar_table */
   else                                             /* is SONARD which is defined in
*/
   {                                                /* sonar.h */
      sonarX = sonar_table[SonarNumber].SonarPosit.X;
      sonarY = sonar_table[SonarNumber].SonarPosit.Y;
      sonarTheta = sonar_table[SonarNumber].SonarTheta;

      robotX = sonar_table[SonarNumber].posit.X;
      robotY = sonar_table[SonarNumber].posit.Y;
      robotTheta = sonar_table[SonarNumber].t;

                     /* vehicle compose sonar */
                     /* --------------------- */

      /* global sonar config = global robot config o local sonar config */
      /* --------------------------------------------------------------- */

/* | lx | |robotX     | |cos(robotTheta)  -sin(robotTheta)  0| |sonarX     |
   | ly |=|robotY     |+|sin(robotTheta)   cos(robotTheta)  0|*|sonarY     |
   | lt | |robotTheta| |       0                  0         1| |sonarTheta|
   |    | |          | |                                     | |          | */


      lx = robotX + (cos(robotTheta) * sonarX) - (sonarY * sin(robotTheta));
      ly = robotY + (sin(robotTheta) * sonarX) + (sonarY * cos(robotTheta));
      lt = sonarTheta + robotTheta;


                     /* vehicle compose sonar range */
                     /* --------------------------- */
```

```
      sonar_table[SonarNumber].global.X = lx + (cos(lt) * range);
      sonar_table[SonarNumber].global.Y = ly + (sin(lt) * range);

  }


  LogSonarData(SonarNumber);

  EnableInterrupts();

}

/********************* Source Code For Linear Fitting ************************/
/*                     ==============================                        */

/*****************************************************************************
 FUNCTION      : SetSonarParameters(double c1, double c2)
 PARAMETERS    : double c1
               : double c2
 PURPOSE       : It allows the user to adjust constants which control the
               : linear fitting algorithm.
               : c1 is a multiplier to allow more lenancy for greater sonar
               : ranges.  c2 is an absolute value.  c1 and c2 are used to
               : determine if an individual data point is usable for the
               : algorithm.
               : Default values are set in main.c to 0.02, 5.0 respectively.
 RETURNS       : void
 CALLS         : None
 CALLED BY     : InitSonar() in sonar.h.
 COMMENTS      : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
SetSonarParameters(double c1, double c2)
{
   C1 = c1;
   C2 = c2;
}


/*****************************************************************************
 FUNCTION      : EnableLinearFitting(int SonarNumber)
 PARAMETERS    : int   SonarNumber
 PURPOSE       : It causes the background system to gather data points from
               : sonar SonarNumber and form them into line segments as governed
               : by the linear fitting algorithm.
 RETURNS       : None
 CALLS         : None
 CALLED BY     : None
 COMMENTS      : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
EnableLinearFitting(int SonarNumber)
{
   sonar_table[SonarNumber].fitting = 1;  /* sonar_table is global variable
                                             which is defined in sonar.c the
                                             data structure of sonar_table is
                                             SONARD which is defined in sonar.h */
}

/*****************************************************************************
 FUNCTION      : DisableLinearFitting(int SonarNumber)
 PARAMETERS    : int   SonarNumber
 PURPOSE       : It causes background system to cease forming line segments for
               : sonar SonarNumber.
 RETURNS       : None
 CALLS         : GenerateSegment() in sonarmath.h.
 CALLED BY     : None
 COMMENTS      : Cleaned and documented by Mahmoud Wahdan on 3-7-95
```

```
*****************************************************************************/

void
DisableLinearFitting(int SonarNumber)
{
   GenerateSegment(SonarNumber);
   sonar_table[SonarNumber].fitting = 0;

}

/*****************************************************************************
 FUNCTION       : LinearFitting(int SonarNumber)
 PARAMETERS     : int   SonarNumber
 PURPOSE        : It controls the fitting of point data to straight line segments.
                : First, it tests if the new coming point is not far from the
                : fitted line. If the test is passed, the point is added to test
                : if the thinness test is passed. If it is passed, the addition is
                : finalized. If any of the tests fail, the line segment is ended
                : and a new one started. The completed line segment is stored in a
                : data structure called SEGMENT, and segments are linked together
                : in a linked list.
 RETURNS        : None
 CALLS          : GenerateSegment()    in sonarmath.h
                : DisableInterrupts()  in sparc.c
                : EnableInterrupts()   in sparc.c
                : AddToSegment() in sonarmath.c  *Local Prototypes in sonarmath.c*
 CALLED BY      : SonarSysControl() in sonar.h
 COMMENTS       : Revised by Y. Kanayama,07-07-93
                : Cleaned and documented by Mahmoud Wahdan on 3-7-95
                : DisableInterrupts() & EnableInterrupts() added by
                : Mahmoud wahdan on Aug 16,95 to solve the problem of
                : some numbers becomes very big or very somall (wrong calculation).
*****************************************************************************/

void
LinearFitting(int SonarNumber)
{
   double         sonar_range;
   double         sonar_range0;
   POINT          p;
   double         alpha, r, delta;

   DisableInterrupts();

   sonar_range  = sonar_table[SonarNumber].d;
   sonar_range0 = sonar_table[SonarNumber].d0;

   if (sonar_range < 9.3 || sonar_range > 409.0)
   {
      GenerateSegment(SonarNumber);
      EnableInterrupts();
      return;
   }

   p = sonar_table[SonarNumber].global;

   if (fabs(sonar_range - sonar_range0) > 3.0 )
   {
      GenerateSegment(SonarNumber);
      AddToSegment(SonarNumber, p);
      EnableInterrupts();
      return;
   }
    if (segment_data[SonarNumber].m00 < 1.5)
   {
      AddToSegment(SonarNumber, p);
      EnableInterrupts();
      return;
    }

   r = segment_data[SonarNumber].seg.r;
```

```
   alpha = segment_data[SonarNumber].seg.alpha;
   delta = fabs(r - p.X * cos(alpha) - p.Y * sin(alpha));

   if (delta > MAXVAL(C2, C1 * sonar_range))
        GenerateSegment(SonarNumber);

   AddToSegment(SonarNumber, p);
   EnableInterrupts();
   return;

         /* End linear_fitting */
 }

/*****************************************************************************
 FUNCTION      : AddToSegment(int SonarNumber)
 PARAMETERS    : int   SonarNumber
               : POINT p
 PURPOSE       : It calculates new interim data for the line segment and stores
               : it in segment_data[SonarNumber].
               : It also changes the end point values to the point being added.
 RETURNS       : None
 CALLS         : LinearFitting() in sonarmath.h.
 CALLED BY     : SonarSysControl() in sonar.h.
 COMMENTS      : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/
void
AddToSegment(int SonarNumber, POINT p)
{
   double    m00, m10, m01, m20, m11, m02;
   double    alpha, r;
   double    mux, muy, mm20,mm11, mm02;
   double    x, y;


   m00 = segment_data[SonarNumber].m00 += 1.0;
   m10 = segment_data[SonarNumber].m10 += p.X;
   m01 = segment_data[SonarNumber].m01 += p.Y;
   m20 = segment_data[SonarNumber].m20 += SQR(p.X);
   m11 = segment_data[SonarNumber].m11 += p.X * p.Y;
   m02 = segment_data[SonarNumber].m02 += SQR(p.Y);

   if (m00 < 1.5)

      segment_data[SonarNumber].seg.start = p;

   mux = m10 / m00;
   muy = m01 / m00;
   mm20 = m20 - SQR(m10) / m00;
   mm11 = m11 - m10 * m01 / m00;
   mm02 = m02 - SQR(m01) / m00;
   y = -2.0 * mm11;
   x = mm02 - mm20;


   if (m00 > 1.5) {

      if (y == 0.0 && x == 0.0)    /* This test is put because when */
                                   /* atan2(0,0) the system crashes */
         alpha = 0.0;              /* July 24th 95 by mahmoud wahdan */

      else

         alpha = atan2(y, x) / 2.0;


      r = mux * cos(alpha) + muy * sin(alpha);

      segment_data[SonarNumber].seg.alpha = alpha;
      segment_data[SonarNumber].seg.r = r;
      segment_data[SonarNumber].seg.end = p;
```

```
    }
}

/*****************************************************************************
 FUNCTION       : GenerateSegment(int SonarNumber)
 PARAMETERS     : int   SonarNumber
 PURPOSE        : It completes segments at the end of a data run. Necessary
                : because the linear fitting function only terminates a segment
                : based on the data and it has no way of knowing that the user
                : has stopped collecting data.
 RETURNS        : None
 CALLS          : EndSegment() in sonarmath.h.
                : BuildList() in sonarmath.c    *Local Prototypes in sonarmath.c*
                : ResetMoments() in sonarmath.c *Local Prototypes in sonarmath.c*
 CALLED BY      : DisableLinearFitting() in sonarmath.h.
                : LinearFitting() in sonarmath.h.
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
GenerateSegment(int SonarNumber)
{
     SEGMENT_RES     *seg_ptr;


   if (segment_data[SonarNumber].m00 > 10.0)
   {

      seg_ptr = EndSegment(SonarNumber);

      BuildList(seg_ptr, SonarNumber);

   }

   ResetMoments(SonarNumber);
}

/*****************************************************************************
 FUNCTION       : EndSegment(int SonarNumber)
 PARAMETERS     : int SonarNumber
 PURPOSE        : It allocates memory for the segment data structure, loads the
                : correct values into it and returns a pointer to the structure.
 RETURNS        : a pointer to the SEGMENT structure
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
                : and data structure LINE-SEG is removed and added to
                : SEGMENT_RES. Refer to this data structure in sonar.h
*****************************************************************************/

SEGMENT_RES*
EndSegment(int SonarNumber)
{
   SEGMENT_RES        tmpSeg;
   SEGMENT_RES       *seg_ptr;
   double             length;
   double             delta;

   seg_ptr = &segstruct;
   tmpSeg = segment_data[SonarNumber].seg;

   delta = tmpSeg.start.X * cos(tmpSeg.alpha) +
           tmpSeg.start.Y * sin(tmpSeg.alpha) - tmpSeg.r;

   tmpSeg.start.X -= delta * cos(tmpSeg.alpha);
   tmpSeg.start.Y -= delta * sin(tmpSeg.alpha);
   delta = tmpSeg.end.X * cos(tmpSeg.alpha) +
           tmpSeg.end.Y * sin(tmpSeg.alpha) - tmpSeg.r;

   tmpSeg.end.X -= delta * cos(tmpSeg.alpha);
```

```
    tmpSeg.end.Y -= delta * sin(tmpSeg.alpha);
    length = sqrt(SQR(tmpSeg.start.X - tmpSeg.end.X) +
             SQR(tmpSeg.start.Y - tmpSeg.end.Y));

    seg_ptr->start.X     = tmpSeg.start.X;
    seg_ptr->start.Y     = tmpSeg.start.Y;
    seg_ptr->end.X       = tmpSeg.end.X;
    seg_ptr->end.Y       = tmpSeg.end.Y;
    seg_ptr->alpha       = tmpSeg.alpha;
    seg_ptr->r           = tmpSeg.r;
    seg_ptr->length      = length;
    seg_ptr->SonarNumber = SonarNumber;
    seg_ptr->m00         = segment_data[SonarNumber].m00;

    return seg_ptr;

}

/*****************************************************************************
 FUNCTION       : BuildList(SEGMENT_RES *ptr, int SonarNumber)
 PARAMETERS     : int           SonarNumber
                : SEGMENT_RES  *ptr
 PURPOSE        : It accepts a pointer to a segment data structure and a sonar
                : number, and appends the segment structure to the tail of a
                : linked list of structures for that sonar.
 RETURNS        : None
 CALLS          : LogSonarSegmentData() in sonarlog.h.
 CALLED BY      : GenerateSegment() in sonarmath.h.
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
BuildList(SEGMENT_RES *ptr, int SonarNumber)
{

    int             next;

    next = (SegListTail[SonarNumber] < 99) ? ++SegListTail[SonarNumber] : 0;

    seg_list[SonarNumber][next] = *ptr;

    LogSonarSegmentData(SonarNumber, seg_list[SonarNumber][next]);
    LogSonarSegmentDataArray(SonarNumber, seg_list[SonarNumber][next]);

}


/*****************************************************************************
 FUNCTION       : ResetMoments(int SonarNumber)
 PARAMETERS     : int   SonarNumber
 PURPOSE        : It resets the accumulative values in segment_data[SonarNumber]
                : (m00, m10, m01, m20, m11, m02) to zero.
 RETURNS        : None
 CALLS          : None
 CALLED BY      : InitSonarmath() in sonarmath.h.
                : GenerateSegment() in sonarmath.h.
 COMMENTS       : Cleaned and documented by Mahmoud Wahdan on 3-7-95
*****************************************************************************/

void
 ResetMoments(int SonarNumber)
{
    segment_data[SonarNumber].m00 = 0.0;
    segment_data[SonarNumber].m10 = 0.0;
    segment_data[SonarNumber].m01 = 0.0;
    segment_data[SonarNumber].m20 = 0.0;
    segment_data[SonarNumber].m11 = 0.0;
    segment_data[SonarNumber].m02 = 0.0;
}

/*****************************************************************************
```

```
 FUNCTION       : GetLastSegment(int SonarNumber)
 PARAMETERS     : int SonarNumber
 PURPOSE        : It returns a pointer to the latest segment on the linked list
                : of segments for sonar.
                : If GetLastSegment is called on an empty list a null pointer is
                : returned.
 RETURNS        : returns the pointer to the last segment on the linked list of
                : segments for sonar. NULL  if there is empty list
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Written by Mahmoud Wahdan on 7-3-95.
*****************************************************************************/

SEGMENT_RES *
GetLastSegment(int SonarNumber)
{
   SEGMENT_RES     *Last_seg;
   int              index;

   index = SegListTail[SonarNumber];

   if (index == -1)

        Last_seg = NULL;

   else
        Last_seg = &seg_list[SonarNumber][index];

   return Last_seg;
}

/*****************************************************************************
 FUNCTION       : GetCurrentSegment(int SonarNumber)
 PARAMETERS     : int SonarNumber
 PURPOSE        : It returns a pointer to the segment currently under construction
                : if there is one, otherwise, returns null pointer. This is
                : accomplished by calling EndSegment, copying the data into
                : segstruct and then returning a pointer to segstruct. The memory
                : allocated by EndSegment is then freed.
 RETURNS        : returns the pointer to the current segment under construction.
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Written by Mahmoud Wahdan on 7-3-95.
*****************************************************************************/

SEGMENT_RES *
GetCurrentSegment(int SonarNumber)
{
   SEGMENT_RES       *seg_ptr;

   seg_ptr = EndSegment(SonarNumber);

   return seg_ptr;

}

/******************** Source Code For Localization Correction ***************/
/*                      ========================================             */

/*****************************************************************************
 FUNCTION       : GetSegmentConfig()
 PARAMETERS     : None
 PURPOSE        : It returns a configuration after applied the LF. It is used in
                  the localization correction.
 RETURNS        : returns a configuration.
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Written by Mahmoud Wahdan on 10-19-95.
*****************************************************************************/

CONFIGURATION
```

```
GetSegmentConfig()

{
   CONFIGURATION       qsegment;
   double              x, y;

   DisableInterrupts();
   qsegment.Posit.X = seg_end_config->end.X;

   qsegment.Posit.Y = seg_end_config->end.Y;

   x                = seg_end_config->start.X - seg_end_config->end.X;
   y                = seg_end_config->start.Y - seg_end_config->end.Y;


   if (y == 0.0 && x == 0.0)

        qsegment.Theta = 0.0;

   else

        qsegment.Theta = atan2(y, x);

   qsegment.Kappa   = 0.0;
   EnableInterrupts();

   return qsegment;

}

/*****************************************************************************
 FUNCTION       : WaitSegment(int SonarNumber)
 PARAMETERS     : int   SonarNumber
 PURPOSE        : busy wait until the line segment being built is completed.
                  It is used in localization correction.
 RETURNS        : void
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Written by Mahmoud Wahdan on 10-19-95.
*****************************************************************************/
void
WaitSegment(int SonarNumber)

{
   int              index;
   SEGMENT_RES      *seg;

   index = SegListTail[SonarNumber];


   while(index == SegListTail[SonarNumber]) {
      ;
   }

   seg = &seg_list[SonarNumber][SegListTail[SonarNumber]];

   seg_end_config = &seg_list[SonarNumber][SegListTail[SonarNumber]];

}

/*****************************************************************************
 FUNCTION       : Match()
 PARAMETERS     : CONFIGURATION : qnew_seg
                : CONFIGURATION : qmodel_seg
 PURPOSE        : compare between new segment and model wall segment.
                  if match then call localization correction routine.
                  It is used in localization correction.
 RETURNS        : void
 CALLS          : WaitSegment() in sonarmath.c
                : setRobotConfig() in seqcmd.c
                : compose()        in geometry.c
```

```
                : inverse()        in geometry.c
 CALLED BY      : none
 COMMENTS       : written by Mahmoud Wahdan on Nov. 4, 95
*****************************************************************************/
/*
Match(CONFIGURATION qnew_seq, CONFIGURATION qmodel_seq)
{

   double length = 50.0;
   double orintation = 2.0;
   double delta_dist;
   double delta_theta;
   DisableInterrupts();
   delta_dist  = eu_dis(qnew_seq.Posit.X, qnew_seq.Posit.Y,
                        qmodel_seq.Posit.X, qmodel_seq.Posit.Y);
   delta_dist  = sqrt((qnew_seq.Posit.X - qmodel_seq.Posit.X) *
                      (qnew_seq.Posit.X - qmodel_seq.Posit.X) +
                      (qnew_seq.Posit.Y - qmodel_seq.Posit.Y) *
                      (qnew_seq.Posit.Y - qmodel_seq.Posit.Y));

   delta_theta = (qnew_seq.Theta - (qmodel_seq.Theta/RAD));

   if((delta_dist < length/2.0) && (delta_theta < (orintation/RAD)))
       return -1;
   else
       return 0;
 }
*/

/*****************************************************************************
 FUNCTION       : CorrectOdometryError()
 PARAMETERS     : CONFIGURATION : qsonar
                : CONFIGURATION : qmodel
 PURPOSE        : correct the error if there is a difference between where
                : the vehicle thinks it is and where the vehicle really is.
 RETURNS        : void
 CALLS          : getRobotConfig() in motion.c
                : setRobotConfig() in seqcmd.c
                : compose()        in geometry.c
                : inverse()        in geometry.c
 CALLED BY      : none
 COMMENTS       : written by Mahmoud Wahdan on Jul. 30, 95
*****************************************************************************/
/*
void
CorrectOdometryError(CONFIGURATION qsegment, CONFIGURATION qmodel)
{
   CONFIGURATION qodometry;
   CONFIGURATION qsegment_inverse;
   CONFIGURATION qactual;
   CONFIGURATION epsilon;
*/
         /*******************************************************/
         /*  qacual    = qmodel o qsegment_inverse o qodometry  */
         /*            = epsilon o qodometry                    */
         /*  epsilon   = qmodel o qsegment_inverse              */
         /*******************************************************/
/*
   qodometry = getRobotConfig();

     qsegment_inverse = inverse(qsegment);
     epsilon          = compose(&qmodel, &qsegment_inverse);
     qactual          = compose(&epsilon, &qodometry);

   setRobotConfigImm(qactual);

#ifdef LOCALIZATION

IOprintf(localizationIO,"%f %f %f\n\n%f %f %f\n\n%f %f %f\n\n%f %f %f\n\n",
           epsilon.Posit.X,
           epsilon.Posit.Y,
```

```
            epsilon.Theta,

            qsegment.Posit.X,
            qsegment.Posit.Y,
            qsegment.Theta,

            qodometry.Posit.X,
            qodometry.Posit.Y,
            qodometry.Theta,

            qactual.Posit.X,
            qactual.Posit.Y,
            qactual.Theta
            );

#endif

}
*/


/****************** Source Code For Linear Fitting with Decay ***************/
/*                  ==========================================                */

/*****************************************************************************
 FUNCTION      : EnableLinearFittingDecay(int SonarNumber)
 PARAMETERS    : int   SonarNumber
 PURPOSE       : It causes the background system to gather data points from
               : sonar SonarNumber and form them into a continuously updated
               : line segments as governed by the linear fitting with
               : decay algorithm.
 RETURNS       : None
 CALLS         : None
 CALLED BY     : None
 COMMENTS      : Added by Dan Wells on 04-15-99
*****************************************************************************/

void
EnableLinearFittingDecay(int SonarNumber)
{
   sonar_table[SonarNumber].decayFitting = 1;  /* sonar_table is global variable
                                                  which is defined in sonar.c. the
                                                  data structure of sonar_table is
                                               SONARD which is defined in sonar.h */
}


/*****************************************************************************
 FUNCTION      : DisableLinearFittingDecay(int SonarNumber)
 PARAMETERS    : int   SonarNumber
 PURPOSE       : It causes background system to cease linear fitting with decay
               : for sonar SonarNumber.
 RETURNS       : None
 CALLS         : None
 CALLED BY     : None
 COMMENTS      : Added by Dan Wells on 04-15-99
*****************************************************************************/

void
DisableLinearFittingDecay(int SonarNumber)
{
   sonar_table[SonarNumber].decayFitting = 0;

}


/*****************************************************************************
 FUNCTION      : ResetMomentsDecay(int SonarNumber)
 PARAMETERS    : int   SonarNumber
 PURPOSE       : It resets the values in seg_data_decay[SonarNumber].
               : Effectively restarts the linear fitting with decay algorithm.
```

```
               : A static variable is used to test for the initial call and
               : if so, sets the decay factor to the default, otherwise
               : leaves this field unchanged. This allows the user to
               : set the decay factor through the SetDecayFactor() function.
 RETURNS       : None
 CALLS         : None
 CALLED BY     : InitSonarmath() in sonarmath.h.
 COMMENTS      : Added by Dan Wells on 04-15-99.
*****************************************************************************/

void
ResetMomentsDecay(int SonarNumber)
{
   static int firstTimeCalled = 1;

   seg_data_decay[SonarNumber].SonarNumber = SonarNumber;
   seg_data_decay[SonarNumber].usable = 0;
   seg_data_decay[SonarNumber].numPoints = 0.0;
   seg_data_decay[SonarNumber].alpha = PI/2.0;
   seg_data_decay[SonarNumber].r = 0.0;
   if (firstTimeCalled) {
      seg_data_decay[SonarNumber].decayFactor = DEFAULT_DECAY;
      firstTimeCalled = 0;
   }
   seg_data_decay[SonarNumber].m00 = 0.0;
   seg_data_decay[SonarNumber].m10 = 0.0;
   seg_data_decay[SonarNumber].m01 = 0.0;
   seg_data_decay[SonarNumber].m11 = 0.0;
   seg_data_decay[SonarNumber].m20 = 0.0;
   seg_data_decay[SonarNumber].m02 = 0.0;
}


/*****************************************************************************
 FUNCTION      : LinearFittingDecay(int SonarNumber)
 PARAMETERS    : int   SonarNumber
 PURPOSE       : It controls the fitting of point data to the current linear
               : fitted line using a decay factor for previously entered
               : points.
 RETURNS       : None
 CALLS         : DisableInterrupts()  in sparc.c
               : EnableInterrupts()   in sparc.c
               : AddToSegmentDecay() in sonarmath.c  *Local Prototypes in sonarmath.c*
 CALLED BY     : SonarSysControl() in sonar.h
 COMMENTS      : Added by Dan Wells on 04-15-99
               : DisableInterrupts() & EnableInterrupts() added to solve the
               : problem of some numbers becomes very big or very
               : small (wrong calculation).
*****************************************************************************/

void
LinearFittingDecay(int SonarNumber)
{
   double         sonar_range;
   double         sonar_range0;
   POINT          p;

   DisableInterrupts();

   sonar_range  = sonar_table[SonarNumber].d;
   sonar_range0 = sonar_table[SonarNumber].d0;

   if (sonar_range < 9.3 || sonar_range > 80) /****409.0)*****/
   {
      EnableInterrupts();
      return;
   }

   p = sonar_table[SonarNumber].global;

   AddToSegmentDecay(SonarNumber, p);
```

```
   EnableInterrupts();
   return;

 }

/*****************************************************************************
 FUNCTION      : AddToSegmentDecay(int SonarNumber, POINT p)
 PARAMETERS    : int   SonarNumber
               : POINT p
 PURPOSE       : It calculates new line segment using linear fitting with decay.
 RETURNS       : None
 CALLS         : None
 CALLED BY     : LinearFittingDecay() in sonarmath.h.
 COMMENTS      : Added by Dan Wells 04-15-99.
*****************************************************************************/
void
AddToSegmentDecay(int SonarNumber, POINT p)
{
   double    decay = seg_data_decay[SonarNumber].decayFactor;
   double    m00, m10, m01, m20, m11, m02;
   double    alpha, r;
   double    mux, muy, M20,M11, M02;
   double    x, y;


   m00 = seg_data_decay[SonarNumber].m00;
   m10 = seg_data_decay[SonarNumber].m10;
   m01 = seg_data_decay[SonarNumber].m01;
   m20 = seg_data_decay[SonarNumber].m20;
   m11 = seg_data_decay[SonarNumber].m11;
   m02 = seg_data_decay[SonarNumber].m02;

   m00 = m00 * decay + 1.0;
   m10 = m10 * decay + p.X;
   m01 = m01 * decay + p.Y;
   m11 = m11 * decay + p.X * p.Y;
   m20 = m20 * decay + p.X * p.X;
   m02 = m02 * decay + p.Y * p.Y;

   mux = m10 / m00;
   muy = m01 / m00;
   M11 = m11 - (m10 * m01) / m00;
   M20 = m20 - SQR(m10) / m00;
   M02 = m02 - SQR(m01) / m00;

   seg_data_decay[SonarNumber].m00 = m00;
   seg_data_decay[SonarNumber].m10 = m10;
   seg_data_decay[SonarNumber].m01 = m01;
   seg_data_decay[SonarNumber].m11 = m11;
   seg_data_decay[SonarNumber].m20 = m20;
   seg_data_decay[SonarNumber].m02 = m02;

   seg_data_decay[SonarNumber].numPoints += 1.0;

   if (seg_data_decay[SonarNumber].numPoints > 1.5) {  /* Need 2 data  */
                                                       /* for a line   */
      seg_data_decay[SonarNumber].usable = 1;
      x = M02 - M20;
      y = -2.0 * M11;

      if ((x == 0.0) && (y == 0.0)) {
         alpha = 0.0;
      }
      else {
         alpha = atan2(y, x) / 2.0;
      }

      r = mux * cos(alpha) + muy * sin(alpha);

      seg_data_decay[SonarNumber].alpha = alpha;
      seg_data_decay[SonarNumber].r = r;
```

```
    }
}

/*****************************************************************************
 FUNCTION       : GetSegmentDecay(int SonarNumber)
 PARAMETERS     : int SonarNumber
 PURPOSE        : It returns a pointer to linear fitted with decay segement
                : for the specified sonar.
 RETURNS        : returns the pointer
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Added by Dan Wells on 04-15-99
*****************************************************************************/

SEGMENT_RES_DECAY
GetSegmentDecay(int SonarNumber)
{
   SEGMENT_RES_DECAY tempSegDecay;

   tempSegDecay = seg_data_decay[SonarNumber];
   return tempSegDecay;
}


/*****************************************************************************
 FUNCTION       : SetDecayFactor(int SonarNumber, double decay)
 PARAMETERS     : int SonarNumber
          : double decay
 PURPOSE        : It sets the decay factor for linear fitted with decay
                : algorithm for the specified sonar. It checks decay value
                : between 0 and 1 and sets to default if out of bounds.
 RETURNS        : void
 CALLS          : None
 CALLED BY      : None
 COMMENTS       : Added by Dan Wells on 04-22-99
*****************************************************************************/

void
SetDecayFactor(int SonarNumber, double decay)
{
   if ( (decay < 0.0) || (decay > 1.0) ) {
      printf("\nDecay set to default");
      decay = DEFAULT_DECAY;
   }

   seg_data_decay[SonarNumber].decayFactor = decay;

}
```

APPENDIX B: USER FILE AND NEW WALL COMMAND FILE

```
/*****************************************************************************
  File name:  user.c
  Descriptions:  set up system with initial parameters and motion mode, then
                 check for end of run conditions
  Revision history:
*****************************************************************************/

#include "user.h"
#include "seqcmd.h"
#include "wallcmd.h"
#include "math.h"
#include "constants.h"
#include "queue.h"
#include "geometry.h"

/*****************************************************************************
  Function  : user()
  Purpose   : Makes a complete circuit (returns to start point)
            : using trackWall motion mode
  Parameters: void
  Returns   : void
  Comments  : Dan Wells 5/99
*****************************************************************************/
void user()
{
  CONFIGURATION startConfig, currentConfig;
  int sonarNum, wallSide;
  double sigma, decay, offset;
  int done, logSonar;

  /* Get user input and set appropriate parameters */

  printf("\nEnter 1 to log sonar data, 0 otherwise : ");
  logSonar = GetInt();

  printf("\nInput desired offset from center of robot to wall (cm) : ");
  offset = GetReal();

  printf("\nInput desired smoothness (cm) e.g. 20 : ");
  sigma = GetReal();

  setSigmaImm(sigma);

  printf("\nInput decay factor (e.g. 0.9) : ");
  decay = GetReal();

  printf("\nEnter 0 for wall on RIGHT side");
  printf("\nEnter 1 for wall on LEFT side");
  printf("\nChoice> ");
  wallSide = GetInt();

  /* Turn on motion logging */
  MotionLog(0,50,0);

  /* Set up sonar number depending on wall following side */
  switch (wallSide) {
     case 0:
        sonarNum = S090;
        break;
     case 1:
        sonarNum = S270;
        break;
     default:
        printf("\nERROR: Illegal wall selection in user.c");
        exit(1);
     break;
```

```
  }

  /* Set the linear fitting decay factor for appropriate sonar */
  SetDecayFactor(sonarNum, decay);

  /* Turn on sonar logging if user requested */
  if (logSonar == 1) {
     InitSonarlog();
     SonarLog(10, 0, sonarNum, SONAR_GLOBAL);
  }

  /* This sets up the wall following motion mode */
  trackWall(sonarNum, offset);

  /* Save starting configuration for later comparison */
  startConfig = getRobotConfig();

  /* Let robot start to move (lets it get outside the 100 cm */
  /* of the starting point) */
  while (getTotalDistanceImm() < 100) {
    waitMS(50);
    /* do nothing */
  }

  /* Now check if robot has halted or gone around the circuit */
  /* Checking the halt condition allows data download if robot */
  /* halts before complete circuit is made */
  done = 0;
  while ( (VelocityLinear() > 0.0) && (!done) ) {
    waitMS(50);
    currentConfig = getRobotConfig();
    /* stop if within 100 cm and 30 degrees of starting position */
    if ( (distance(startConfig.Posit,currentConfig.Posit) < 100) &&
         (fabs(norm(startConfig.Theta - currentConfig.Theta)) < 30*d2r) ) {
      done = 1;
    }
  }

  /* Disable everything */
  DisableLinearFittingDecay(sonarNum);
  DisableSonar(sonarNum);
  DisableSonar(S000);

  /* Stop robot */
  stopImm();

  return;  /* Returns to main for data download */
}
```

```
/*****************************************************************************
  File name:  wallcmd.c
  Descriptions:  set up the currentPath variable in motion.c for wall following
                 mode
  Revision history:
*****************************************************************************/

#include "definitions.h"
#include "motion.h"
#include "stdiosys.h"
#include <math.h>
#include "geometry.h"
#include "sonar.h"
#include "sonarcard.h"
#include "sonarmath.h"
```

```
/*****************************************************************************
FUNCTION:   trackWall()
PARAMETERS: sonarNum - specifies left/right wall by sonar number S270/S090
            offset - specifies offset distance magnitude from wall
PURPOSE:          sets up the currentPath LINE variable in motion.c for wall
            following
RETURNS:    void
CALLED BY: user()
CALLS:            setCurrentPath(), setRobotConfigImm()
COMMENTS:
*****************************************************************************/
void
trackWall(int sonarNum, double offset)

{
  double k;
  LINE pathElement;
  double sigma = sigmaValue();

  /* sonarNum determines side of wall following          */
  /* Constants defined in sonar.h                       */
  /* S090 - wall on right side, S270 - wall on left side */
  switch (sonarNum) {
     case S090:
        pathElement.Type = TRACKRWALLMODE;
        offset = offset < fabs(sonar_table[S090].SonarPosit.Y)+MIN_EDGE_OFFSET ?
                 DEFAULT_WALL_OFFSET : offset;
        break;
     case S270:
        pathElement.Type = TRACKLWALLMODE;
        offset = offset < fabs(sonar_table[S270].SonarPosit.Y)+MIN_EDGE_OFFSET ?
                 -DEFAULT_WALL_OFFSET : -offset;
        break;
     default:
        printf("\nERROR: illegal value for sonarNum in trackWall()");
        exit(1);
        break;
  }


  pathElement.config = defineConfig(0.0, 0.0, 0.0, 0.0);

  k = 1.0/sigma ;
  pathElement.a = 3.0*k ;
  pathElement.b = 3.0*k*k;
  pathElement.c = k*k*k;
  pathElement.Radius = offset; /* won't initially use Radius in wall following
                                  so we use it to pass in the requested
                                  wall offset */
  pathElement.Center.X = 0.0;
  pathElement.Center.Y = 0.0;

  setPathElement(pathElement); /* defined in motion.c - sets currentPath */
  setRobotConfigImm(pathElement.config); /* matches robot with currentPath */

  EnableSonar(S000);
  EnableSonar(sonarNum);
  EnableLinearFittingDecay(sonarNum);

  return;
}
```

APPENDIX C: THE *WALLHUGRULE* FUNCTION

```
1   /*****************************************************************************
2     Function  : wallHugRule()
3     Purpose   : To determine the linear and rotational velocities needed to put
4                 or keep Yamabico at the correct offset from the wall.
5     Parameters: VELOCITY actual, commanded
6     Returns   : The required linear velocity, rotational velocity
7     Comments:
8   *****************************************************************************/
9   static  VELOCITY
10  wallHugRule(VELOCITY actual, VELOCITY commanded)
11  {
12
13    CONFIGURATION curWall, offsetConfig, relativeConfig, inverseConfig,
14                  tempConfig;
15    SEGMENT_RES_DECAY  curSeg;
16    double wallx, wally, wallTheta;
17    static double prevWallTheta=0.0;
18
19    static double initCircleTheta;
20
21    double deltaDistance;
22    double deltaAngle;
23    static double savedSigma; /* holds old sigma value while in turn */
24    static int pingedOnce = 0,
25               pingedTwice = 0;
26    static int firstTimeCalled = 1; /*only true if first time thru wallHugRule*/
27
28    int sonarNum;
29    double dist000, sideDist, lastSideDist; /* sonar distances */
30
31    static double offset; /* offset is stored in the initially unused */
32                  /* (for this mode) currentPath.Radius field */
33                  /* offset < 0 for left wall following       */
34                  /* offset > 0 for right wall following      */
35
36    /* flags to control motion calculation */
37    static int halted = 0;    /* indicates motion halted (collision) */
38    static int inTurn = 0;    /* designates when robot in a turn */
39    static int inTransition = 0; /* within transition area but still  */
40                                 /* waiting for usable linfit data    */
41    static int turnType = 0; /* 0=inside turn, 1=outside turn      */
42                             /* 2=on circle */
43    int trackType;           /* 0=wall follow, 1=track currentPath */
44
45    double jL;  /* Steering function value */
46
47    double kk, kk2;     /* kappa, kappa squared */
48    double k;           /* steering gain parameter */
49
50    /* Set sonarNum, turn angle, and offset for left or right wall following */
51    /* Note: turn angle (deltaAngle) is set for inside turns, use negative   */
52    /* of this value for outside turns                                       */
53    switch (currentPath.Type) {
54       case TRACKRWALLMODE:
55          sonarNum = S090;
56          deltaAngle = PI/2.0;
57          break;
58       case TRACKLWALLMODE:
59          sonarNum = S270;
60          deltaAngle = - PI/2.0;
61          break;
62       default:
63          printf("\nERROR in wallHugRule()");
64          exit(1);
65          break;
66    }
```

```
67
68    /* offset initially stored in currentPath.Radius */
69    if (firstTimeCalled) {
70       offset = currentPath.Radius;
71    }
72
73    /* Get current configuation of the linear fitted (w/ decay) wall */
74    curSeg = GetSegmentDecay(sonarNum);
75
76    /* calculate the wall angle */
77    if (fabs(norm((curSeg.alpha - PI/2.0) - vehicle.Theta)) < PI/2.0) {
78      wallTheta = norm(curSeg.alpha - PI/2.0);
79    }
80    else {
81      wallTheta = norm(curSeg.alpha + PI/2.0);
82    }
83
84
85    /* first check forward collision eminent */
86    if (halted ||
87       ((sonar_table[S000].update) &&
88        (sonar_table[S000].d < COLLISION_THRESH))) {
89      halted = 1;
90      commanded.Linear = 0.0;
91      commanded.Rotational = 0.0;
92      return commanded;
93    }
94
95    /* ensure good side sonar history data */
96    if (!pingedTwice && sonar_table[sonarNum].update) {
97      sonar_table[sonarNum].update = 0;
98      if (pingedOnce) {
99        pingedTwice = 1;
100     }
101     else {
102       pingedOnce = 1;
103     }
104   }
105
106   /*-------------------------------------------------------------*/
107   /* Handle all phases: wall following, time to turn, in turn, */
108   /* in transition, return to wall following from turn          */
109   /*-------------------------------------------------------------*/
110
111   /* not in turn (following wall), check for turn conditions */
112   if (!inTurn) {
113
114     /* set up sonar distances used in checks */
115     dist000 = sonar_table[S000].d + sonar_table[S000].SonarPosit.X;
116     sideDist = sonar_table[sonarNum].d;
117     lastSideDist = sonar_table[sonarNum].d0;
118
119     /* check distance from front wall, if past neutral switching point */
120     /* start inside turn to new path */
121     if ( (sonar_table[S000].update) &&
122          (dist000 <= 3.0*INSIDE_TURN_SIGMA*PI/2.0 + fabs(offset)) ) {
123       inTurn = 1;
124       turnType = 0;
125       savedSigma = sigmaValue();
126       setSigmaImm(INSIDE_TURN_SIGMA);
127       offsetConfig = defineConfig(dist000-fabs(offset), 0.0, deltaAngle, 0.0);
128
129       /* set current path to approximate next wall offset path */
130       currentPath.config = compose(&vehicle, &offsetConfig);
131       currentPath.config.Kappa = 0.0;
132       k = 1.0/sigmaValue();
133       currentPath.a = 3.0*k ;
134       currentPath.b = 3.0*k*k;
135       currentPath.c = k*k*k;
136
137       trackType = 1;
```

```
138       sonar_table[S000].update = 0;
139       DisableSonar(sonarNum);
140     }
141
142     /* check for an outside turn, indicated by large difference between */
143     /* current and previous side sonar measurements                     */
144     else if ( (sonar_table[sonarNum].update) &&
145               (pingedTwice) &&
146               (fabs(sideDist - lastSideDist) > OUTSIDE_TURN_THRESH) ) {
147       inTurn = 1;
148       savedSigma = sigmaValue();
149       setSigmaImm(OUTSIDE_TURN_SIGMA);
150       turnType = 2;
151
152       /* make current path a circle around the corner */
153       /* The radius was changed from the offset value to */
154       /* 3/4 offset value for a little tighter turn */
155       initCircleTheta = norm(vehicle.Theta); /* save for later comparison */
156       kk = -1.0/(offset*3.0/4.0);
157       kk2 = kk * kk;
158       k = 1.0/sigmaValue();
159       currentPath.a = 3.0*k ;
160       currentPath.b = 3.0*k*k - kk2 ;
161       currentPath.c = k*k*k - 3.0*k*kk2 ;
162       currentPath.config = defineConfig(vehicle.Posit.X, vehicle.Posit.Y,
163                                         initCircleTheta, kk);
164
165       currentPath.Radius = 1.0/kk;
166       offsetConfig = defineConfig(0.0, -offset, 0.0, 0.0);
167       tempConfig = compose(&currentPath.config, &offsetConfig);
168       currentPath.Center.X = tempConfig.Posit.X;
169       currentPath.Center.Y = tempConfig.Posit.Y;
170
171       trackType = 1;
172       sonar_table[sonarNum].update = 0;
173       DisableSonar(sonarNum);
174     }
175
176     /* otherwise keep following the wall */
177     else {
178       trackType = 0;
179     }
180   }
181
182   /* in a turn, check for transition criteria */
183   else if (!inTransition) {
184
185     /* check to see if within transition area now, reset linfit data if so */
186
187     /* check for inside turn, vehicle angle close to line angle */
188     if ( (turnType == 0) &&
189          (fabs(norm(vehicle.Theta - currentPath.config.Theta)) <
190                                                          TRANSITION_ANGLE)){
191       inTransition = 1;
192       EnableSonar(sonarNum);
193       ResetMomentsDecay(sonarNum);
194       pingedOnce = pingedTwice = 0;
195     }
196     /* for outside turn check if past the corner */
197     else if (turnType == 1) {
198       inverseConfig = inverse(currentPath.config);
199       relativeConfig = compose(&inverseConfig, &vehicle);
200       /* check if past the wall corner */
201       if (relativeConfig.Posit.X >  25) {
202         inTransition = 1;
203         EnableSonar(sonarNum);
204         ResetMomentsDecay(sonarNum);
205         pingedOnce = pingedTwice = 0;
206       }
207     }
208     /* if still on circle, check to see if angle has changed by */
```

```
209     /* 90 +/- 3 degrees from the last wall or the initial circle angle, */
210     /* whichever comes first */
211     else if ( (turnType == 2) &&
212               ((fabs(norm(vehicle.Theta - prevWallTheta + deltaAngle)) <=
213                                                                   3*d2r) ||
214               (fabs(norm(vehicle.Theta - initCircleTheta + deltaAngle)) <=
215                                                                   3*d2r))) {
216       /* proceed on a line straight ahead */
217       currentPath.config = defineConfig(vehicle.Posit.X, vehicle.Posit.Y,
218                                         vehicle.Theta, 0.0);
219       k = 1.0/sigmaValue();
220       currentPath.a = 3.0*k ;
221       currentPath.b = 3.0*k*k;
222       currentPath.c = k*k*k;
223
224       turnType = 1;
225     }
226
227
228     /* keep tracking toward new path line */
229     trackType = 1;
230   }
231
232   /* in transition area */
233   else {
234
235     /* check to see if new linfit data is usable, return to wall following */
236     if (curSeg.usable) {
237       inTransition = 0;
238       inTurn = 0;
239       trackType = 0;
240       setSigmaImm(savedSigma);
241     }
242
243     /* otherwise keep tracking toward new path */
244     else {
245       trackType = 1;
246     }
247   }
248
249   switch (trackType) {
250     /* follow wall represented by linear fitted sonar data */
251     case 0:
252       wallx = curSeg.r * cos(curSeg.alpha);
253       wally = curSeg.r * sin(curSeg.alpha);
254       /* wallTheta defined earlier */
255       curWall = defineConfig(wallx, wally, wallTheta, 0.0);
256
257
258       /* Update the current path the robot should be following */
259       offsetConfig = defineConfig(0.0, offset, 0.0, 0.0);
260       currentPath.config = compose(&curWall, &offsetConfig);
261
262       /* Get steering value and update commanded velocities */
263       jL = steerByWall(vehicle, curWall, offset);
264
265       /* save wall angle */
266       prevWallTheta=wallTheta;
267       break;
268
269     /* use normal steering function to track current path */
270     case 1:
271       jL = steer(vehicle, currentPath);
272       break;
273
274     default:
275       printf("\nERROR in wallHugRule()! Exiting...");
276       exit(1);
277       break;
278     }
279
```

```
280   firstTimeCalled = 0;
281
282   commanded.Linear = computeLinVelocity(actual.Linear,commanded.Linear);
283
284   deltaDistance = MOTION_CONTROL_CYCLE * commanded.Linear;
285
286   kappaCommanded = vehicle.Kappa + jL * deltaDistance;
287   commanded.Rotational = kappaCommanded * commanded.Linear;
288
289   return commanded;
290 }
```

LIST OF REFERENCES

[1] Naval Postgraduate School Monterey CA, *Yamabico User's Manual*, October 1994.

[2] Kanayama, Yutaka, "Introduction to Two-Dimensional Robotics," *Lecture Notes for CS4314*, Department of Computer Science, Naval Postgraduate School, March 1998.

[3] Kanayama, Yutaka, "Two Dimensional Wheeled Vehicle Kinematics," *Proc. IEEE International Conference on Robotics and Automation*, San Diego, CA, May 8-13, 1994, pp. 3079-3084.

[4] Kanayama, Y. and Krahn, G., "Theory of Two-Dimensional Transformations," *IEEE Transactions on Robotics and Automation*, Vol. 14, No. 5, pp. 827-834.

[5] Kanayama, Y. and Fahroo, F., "A New Line Tracking Method for Nonholonomic Vehicles," *Proc. IEEE International Conference on Robotics and Automation*, Albuqueque, NM, April, 1997, pp. 2908-2913.

[6] Kanayama, Y. and Fahroo, F., "A Circle Tracking Method for Nonholonomic Vehicles," *Proc. The Fifth IFAC Simposium on Robot Control*, Nontes, France, September, 1997, pp. 551-558.

[7] Walpole, R. and Myers, R., *Probability and Statistics for Engineers and Scientists*, 3rd ed., Macmillan, Inc., New York, NY, 1985.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Engineering & Technology Curricular Office, Code 34 1
   Naval Postgraduate School
   Monterey, California 93943

4. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943

5. Dr Yutaka Kanayama, Code CS/KA 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943

6. Professor Thomas Hofler, Code PH/HF 1
   Physics Department
   Naval Postgraduate School
   Monterey, California 93943

7. Professor Thomas Wu, Code CS/WQ 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943

8. Daniel Wells 2
   22 Forest Park Rd.
   Cedar Crest, New Mexico 87008